# C# Tutorial

# C# TUTORIAL

*Simply Easy Learning by tutorialspoint.com*

tutorialspoint.com

# ABOUT THE TUTORIAL

## C# Tutorial

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg.

This tutorial will teach you basic C# programming and will also take you through various advanced concepts related to C# programming language.

## Audience

This tutorial has been prepared for the beginners to help them understand basic C# programming. After completing this tutorial, you will find yourself at a moderate level of expertise in C# programming from where you can take yourself to next levels.

## Prerequisites

C# programming is very much based on C and C++ programming languages, so if you have basic understanding on C or C++ programming, then it will be a fun to learn C# programming language.

## Copyright & Disclaimer Notice



This tutorial may contain inaccuracies or errors and tutorialspoint provides no guarantee regarding the accuracy of the site or its contents including this tutorial. If you discover that the tutorialspoint.com site or this tutorial content contains some errors, please contact us at webmaster@tutorialspoint.com

# Table of Content

CHAPTER 1

# C# Overview

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by Ecma and ISO.

C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.

C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages to be used on different computer platforms and architectures.

The following reasons make C# a widely used professional language:

- Modern, general-purpose programming language
- Object oriented.
- Component oriented.
- Easy to learn.
- Structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- Part of .Net Framework.

## Strong Programming Features of C#

Although C# constructs closely follow traditional high-level languages, C and C++ and being an object-oriented programming language, it has strong resemblance with Java, it has numerous strong programming features that make it endearing to multitude of programmers worldwide.

Following is the list of few important features:

- Boolean Conditions

- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy-to-use Generics
- Indexers
- Conditional Compilation
- Simple Multithreading
- LINQ and Lambda Expressions
- Integration with Windows

CHAPTER 2

# C# Environment

In this chapter, we will discuss the tools required for creating C# programming. We have already mentioned that C# is part of .Net framework and is used for writing .Net applications. Therefore, before discussing the available tools for running a C# program, let us understand how C# relates to the .Net framework.

## The .Net Framework

The .Net framework is a revolutionary platform that helps you to write the following types of applications:

- Windows applications
- Web applications
- Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages like C#. Following are some of the components of the .Net framework:

- Common Language Runtime (CLR)
- The .Net Framework Class Library
- Common Language Specification
- Common Type System
- Metadata and Assemblies
- Windows Forms
- ASP.Net and ASP.Net AJAX
- ADO.Net

- Windows Workflow Foundation (WF)
- Windows Presentation Foundation
- Windows Communication Foundation (WCF)
- LINQ

For the jobs each of these components perform, please see ASP.Net - Introduction, and for details of each component, please consult Microsoft's documentation.

# Integrated Development Environment (IDE) For C#

Microsoft provides the following development tools for C# programming:

- Visual Studio 2010 (VS)
- Visual C# 2010 Express (VCE)
- Visual Web Developer

The last two are freely available from Microsoft official website. Using these tools, you can write all kinds of C# programs from simple command-line applications to more complex applications. You can also write C# source code files using a basic text editor, like Notepad, and compile the code into assemblies using the command-line compiler, which is again a part of the .NET Framework.

Visual C# Express and Visual Web Developer Express edition are trimmed down versions of Visual Studio and has the same look and feel. They retain most features of Visual Studio. In this tutorial, we have used Visual C# 2010 Express.

You can download it from Microsoft Visual Studio. It gets automatically installed in your machine. Please note that you need an active internet connection for installing the express edition.

# Writing C# Programs on Linux or Mac OS

Although the.NET Framework runs on the Windows operating system, there are some alternative versions that work on other operating systems. **Mono** is an open-source version of the .NET Framework which includes a C# compiler and runs on several operating systems, including various flavors of Linux and Mac OS. Kindly check Go Mono.

The stated purpose of Mono is not only to be able to run Microsoft .NET applications cross-platform, but also to bring better development tools to Linux developers. Mono can be run on many operating systems including Android, BSD, iOS, Linux, OS X, Windows, Solaris and UNIX.

CHAPTER

3

# C# Program Structure

Before we study basic building blocks of the C# programming language, let us look at a bare minimum C# program structure so that we can take it as a reference in upcoming chapters.

## C# Hello World Example

A C# program basically consists of the following parts:

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
using System;
namespace HelloWorldApplication
{
   class HelloWorld
   {
      static void Main(string[] args)
      {
         /* my first program in C# */
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello World
```

Let us look at various parts of the above program:

- The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.
- The next line has the **namespace** declaration. A **namespace** is a collection of classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.
- The next line has a **class** declaration, the class *HelloWorld* contains the data and method definitions that your program uses. Classes generally would contain more than one method. Methods define the behavior of the class. However, the *HelloWorld* class has only one method **Main**.
- The next line defines the **Main** method, which is the **entry point** for all C# programs. The **Main** method states what the class will do when executed
- The next line /*...*/ will be ignored by the compiler and it has been put to add additional **comments** in the program.
- The Main method specifies its behavior with the statement **Console.WriteLine("Hello World");**
  *WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
- The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It's worth to note the following points:

- C# is case sensitive.
- All statements and expression must end with a semicolon (;).
- The program execution starts at the Main method.
- Unlike Java, file name could be different from the class name.

# Compile & Execute a C# Program:

If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps:

- Start Visual Studio.
- On the menu bar, choose File, New, Project.
- Choose Visual C# from templates, and then choose Windows.
- Choose Console Application.
- Specify a name for your project, and then choose the OK button.
- The new project appears in Solution Explorer.
- Write code in the Code Editor.
- Click the Run button or the F5 key to run the project. A Command Prompt window appears that contains the line Hello World.

You can compile a C# program by using the command-line instead of the Visual Studio IDE:

- Open a text editor and add the above-mentioned code.
- Save the file as **helloworld.cs**
- Open the command prompt tool and go to the directory where you saved the file.
- Type **csc helloworld.cs** and press enter to compile your code.
- If there are no errors in your code, the command prompt will take you to the next line and would generate **helloworld.exe** executable file.
- Next, type **helloworld** to execute your program.
- You will be able to see "Hello World" printed on the screen.

CHAPTER

4

# C# Basic Syntax

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and display details.

Let us look at an implementation of a Rectangle class and discuss C# basic syntax, on the basis of our observations in it:

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        // member variables
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
```

```
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

# The *using* Keyword

The first statement in any C# program is

```
using System;
```

The **using** keyword is used for including the namespaces in the program. A program can include multiple using statements.

# The *class* Keyword

The **class** keyword is used for declaring a class.

# Comments in C#

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:

```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

Single-line comments are indicated by the '//' symbol. For example,

```
}//end class Rectangle
```

# Member Variables

Variables are attributes or data members of a class, used for storing data. In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

# Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class Rectangle contains three member functions: *AcceptDetails*, *GetArea* and *Display*.

# Instantiating a Class

In the preceding program, the class *ExecuteRectangle* is used as a class, which contains the *Main()* method and instantiates the *Rectangle* class.

# Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.

- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore ( _ ) can be used.

- It should not be a C# keyword.

# C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers; however, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

In C#, some identifiers have special meaning in context of code, such as get and set, these are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C#:

| **Reserved Keywords** | | | | | | |
|---|---|---|---|---|---|---|
| abstract | As | Base | bool | break | byte | case |
| catch | Char | checked | class | const | continue | decimal |
| default | delegate | do | double | else | enum | event |
| explicit | extern | false | finally | fixed | float | for |
| foreach | Goto | if | implicit | in | in (generic modifier) | int |
| interface | internal | is | lock | long | namespace | new |
| null | object | operator | out | out (generic modifier) | override | params |
| private | protected | public | readonly | ref | return | sbyte |
| sealed | short | sizeof | stackalloc | static | string | struct |
| switch | This | throw | true | try | typeof | uint |
| ulong | unchecked | unsafe | ushort | using | virtual | void |
| volatile | While | | | | | |
| **Contextual Keywords** | | | | | | |
| add | Alias | ascending | descending | dynamic | from | get |

| | | | | | | |
|---|---|---|---|---|---|---|
| global | group | into | join | let | orderby | partial (type) |
| partial (method) | remove | select | set | | | |

CHAPTER

5

# C# Data Types

In C#, variables are categorized into the following types:

- Value types
- Reference types
- Pointer types

## Value Types

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.
The value types directly contain data. Some examples are **int, char, float**, which stores numbers, alphabets, floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010:

| Type | Represents | Range | Default Value |
|---|---|---|---|
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | $(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0 \text{ to } 28}$ | 0.0M |
| double | 64-bit double-precision floating point type | $(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$ | 0.0D |
| float | 32-bit single-precision floating point type | $-3.4 \times 10^{38}$ to $+ 3.4 \times 10^{38}$ | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -923,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |

| | | | |
|---|---|---|---|
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine:

```
namespace DataTypeApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Size of int: {0}", sizeof(int));
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Size of int: 4
```

# Reference Types

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using more than one variable, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic** and **string**.

## OBJECT TYPE

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. So object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
object obj;
obj = 100; // this is boxing
```

## DYNAMIC TYPE

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at runtime.

Syntax for declaring a dynamic type is:

```
dynamic <variable_name> = value;
```

For example,

```
dynamic d = 20;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run-time.

### STRING TYPE

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

```
String str = "Tutorials Point";
```

A @quoted string literal looks like:

```
@"Tutorials Point";
```

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

## Pointer Types

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as in C or C++.

Syntax for declaring a pointer type is:

```
type* identifier;
```

For example,

```
char* cptr;
int* iptr;
```

We will discuss pointer types in the chapter 'Unsafe Codes'.

CHAPTER

6

# C# Type Conversion

Type conversion is basically type casting or converting one type of data to another type. In C#, type casting has two forms:

- **Implicit type conversion** - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types and conversions from derived classes to base classes.
- **Explicit type conversion** - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion:

```
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // cast double to int.
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();

        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
5673
```

## C# Type Conversion Methods

C# provides the following built-in type conversion methods:

| S.N | Methods & Description |
|---|---|

| | |
|---|---|
| 1 | **ToBoolean**<br>Converts a type to a Boolean value, where possible. |
| 2 | **ToByte**<br>Converts a type to a byte. |
| 3 | **ToChar**<br>Converts a type to a single Unicode character, where possible. |
| 4 | **ToDateTime**<br>Converts a type (integer or string type) to date-time structures. |
| 5 | **ToDecimal**<br>Converts a floating point or integer type to a decimal type. |
| 6 | **ToDouble**<br>Converts a type to a double type. |
| 7 | **ToInt16**<br>Converts a type to a 16-bit integer. |
| 8 | **ToInt32**<br>Converts a type to a 32-bit integer. |
| 9 | **ToInt64**<br>Converts a type to a 64-bit integer. |
| 10 | **ToSbyte**<br>Converts a type to a signed byte type. |
| 11 | **ToSingle**<br>Converts a type to a small floating point number. |
| 12 | **ToString**<br>Converts a type to a string. |
| 13 | **ToType**<br>Converts a type to a specified type. |
| 14 | **ToUInt16**<br>Converts a type to an unsigned int type. |
| 15 | **ToUInt32**<br>Converts a type to an unsigned long type. |
| 16 | **ToUInt64**<br>Converts a type to an unsigned big integer. |

The following example converts various value types to string type:

```
namespace TypeConversionApplication
{
    class StringConversion
    {
        static void Main(string[] args)
        {
            int i = 75;
            float f = 53.005f;
            double d = 2345.7652;
```

```
            bool b = true;

            Console.WriteLine(i.ToString());
            Console.WriteLine(f.ToString());
            Console.WriteLine(d.ToString());
            Console.WriteLine(b.ToString());
            Console.ReadKey();

        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
75
53.005
2345.7652
True
```

CHAPTER

7

# C# Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

We have already discussed various data types. The basic value types provided in C# can be categorized as:

| Type | Example |
|---|---|
| Integral types | sbyte, byte, short, ushort, int, uint, long, ulong and char |
| Floating point types | float and double |
| Decimal types | decimal |
| Boolean types | true or false values, as assigned |
| Nullable types | Nullable data types |

C# also allows defining other value types of variable like **enum** and reference types of variables like **class**, which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

## Variable Definition in C#

Syntax for variable definition in C# is:

```
<data_type> <variable_list>;
```

Here, data_type must be a valid C# data type including char, int, float, double, or any user-defined data type, etc., and variable_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

You can initialize a variable at the time of definition as:

```
int i = 100;
```

# Variable Initialization in C#

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

```
variable_name = value;
```

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as:

```
<data_type> <variable_name> = value;
```

Some examples are:

```
int d = 3, f = 5;     /* initializing d and f. */
byte z = 22;          /* initializes z. */
double pi = 3.14159; /* declares an approximation of pi. */
char x = 'x';         /* the variable x has the value 'x'. */
```

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

Try the following example, which makes use of various types of variables:

```
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b ;
            double c;

            /* actual initialization */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
a = 10, b = 20, c = 30
```

# Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

```
int num;
num = Convert.ToInt32(Console.ReadLIne());
```

The function **Convert.ToInt32()** converts the data entered by the user to int data type, because **Console.ReadLine()** accepts the data in string format.

## Lvalues and Rvalues in C#:

There are two kinds of expressions in C#:

1. **lvalue:** An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. **rvalue:** An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

```
int g = 20;
```

But following is not a valid statement and would generate compile-time error:

```
10 = 20;
```

CHAPTER

8

# C# Constants and Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

## Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and no prefix is required for decimal numbers.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of Integer literals:

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

## Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Here are some examples of floating-point literals:

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

# Character Constants

Character literals are enclosed in single quotes, e.g., 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C# when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

| Escape sequence | Meaning |
|---|---|
| \\ | \ character |
| \' | ' character |
| \" | " character |
| \? | ? character |
| \a | Alert or bell |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \ooo | Octal number of one to three digits |
| \xhh . . . | Hexadecimal number of one or more digits |

Following is the example to show few escape sequence characters:

```
namespace EscapeChar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello\tWorld\n\n");
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello   World
```

# String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

# Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is:

```
const <data_type> <constant_name> = value;
```

The following program demonstrates defining and using a constant in your program:

```
using System;

namespace DeclaringConstants
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159; // constant declaration
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

CHAPTER

9

# C# Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# is rich in built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This tutorial will explain the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

## Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

| Operator | Description | Example |
| --- | --- | --- |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| * | Multiplies both operands | A * B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | Increment operator increases integer value by one | A++ will give 11 |
| -- | Decrement operator decreases integer value by one | A-- will give 9 |

## Example

Try the following example to understand all the arithmetic operators available in C#:

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int b = 10;
            int c;

            c = a + b;
            Console.WriteLine("Line 1 - Value of c is {0}", c);
            c = a - b;
            Console.WriteLine("Line 2 - Value of c is {0}", c);
            c = a * b;
            Console.WriteLine("Line 3 - Value of c is {0}", c);
            c = a / b;
            Console.WriteLine("Line 4 - Value of c is {0}", c);
            c = a % b;
            Console.WriteLine("Line 5 - Value of c is {0}", c);
            c = a++;
            Console.WriteLine("Line 6 - Value of c is {0}", c);
            c = a--;
            Console.WriteLine("Line 7 - Value of c is {0}", c);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

# Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

| Operator | Description | Example |
|---|---|---|
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |

| | | |
|---|---|---|
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## Example

Try the following example to understand all the relational operators available in C#:

```
using System;

class Program
{
  static void Main(string[] args)
  {
      int a = 21;
      int b = 10;

      if (a == b)
      {
          Console.WriteLine("Line 1 - a is equal to b");
      }
      else
      {
          Console.WriteLine("Line 1 - a is not equal to b");
      }
      if (a < b)
      {
          Console.WriteLine("Line 2 - a is less than b");
      }
      else
      {
          Console.WriteLine("Line 2 - a is not less than b");
      }
      if (a > b)
      {
          Console.WriteLine("Line 3 - a is greater than b");
      }
      else
      {
          Console.WriteLine("Line 3 - a is not greater than b");
      }
      /* Lets change value of a and b */
      a = 5;
      b = 20;
      if (a <= b)
      {
         Console.WriteLine("Line 4 - a is either less than or equal to  b");
      }
      if (b >= a)
      {
         Console.WriteLine("Line 5-b is either greater than or equal to b");
      }
  }
```

```
}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b
```

# Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non zero then condition becomes true. | (A && B) is false. |
| \|\| | Called Logical OR Operator. If any of the two operands is non zero then condition becomes true. | (A \|\| B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

## Example

Try the following example to understand all the logical operators available in C#:

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            bool a = true;
            bool b = true;

            if (a && b)
            {
               Console.WriteLine("Line 1 - Condition is true");
            }
            if (a || b)
            {
                Console.WriteLine("Line 2 - Condition is true");
            }
            /* lets change the value of  a and b */
            a = false;
            b = true;
            if (a && b)
            {
                Console.WriteLine("Line 3 - Condition is true");
            }
            else
            {
                Console.WriteLine("Line 3 - Condition is not true");
```

```
            }
            if (!(a && b))
            {
                Console.WriteLine("Line 4 - Condition is true");
            }
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

# Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. The truth tables for &, |, and ^ are as follows:

| P | Q | p & q | p \| q | p ^ q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12, which is 0000 1100 |

| | | |
|---|---|---|
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -60, which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240, which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15, which is 0000 1111 |

## Example

Try the following example to understand all the bitwise operators available in C#:

```
using System;
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 60;              /* 60 = 0011 1100 */
            int b = 13;              /* 13 = 0000 1101 */
            int c = 0;

             c = a & b;            /* 12 = 0000 1100 */
             Console.WriteLine("Line 1 - Value of c is {0}", c );

             c = a | b;            /* 61 = 0011 1101 */
             Console.WriteLine("Line 2 - Value of c is {0}", c);

             c = a ^ b;            /* 49 = 0011 0001 */
             Console.WriteLine("Line 3 - Value of c is {0}", c);

             c = ~a;                /*-61 = 1100 0011 */
             Console.WriteLine("Line 4 - Value of c is {0}", c);

             c = a << 2;      /* 240 = 1111 0000 */
             Console.WriteLine("Line 5 - Value of c is {0}", c);

             c = a >> 2;      /* 15 = 0000 1111 */
             Console.WriteLine("Line 6 - Value of c is {0}", c);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - Value of c is 12
```

```
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

# Assignment Operators

There are following assignment operators supported by C#:

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| \|= | bitwise inclusive OR and assignment operator | C \|= 2 is same as C = C \| 2 |

## Example

Try the following example to understand all the assignment operators available in C#:

```
using System;

namespace OperatorsAppl
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            int a = 21;
            int c;

            c = a;
            Console.WriteLine("Line 1 - =  Value of c = {0}", c);

            c += a;
            Console.WriteLine("Line 2 - += Value of c = {0}", c);

            c -= a;
            Console.WriteLine("Line 3 - -=  Value of c = {0}", c);

            c *= a;
            Console.WriteLine("Line 4 - *=  Value of c = {0}", c);

            c /= a;
            Console.WriteLine("Line 5 - /=  Value of c = {0}", c);

            c = 200;
            c %= a;
            Console.WriteLine("Line 6 - %=  Value of c = {0}", c);

            c <<= 2;
            Console.WriteLine("Line 7 - <<=  Value of c = {0}", c);

            c >>= 2;
            Console.WriteLine("Line 8 - >>=  Value of c = {0}", c);

            c &= 2;
            Console.WriteLine("Line 9 - &=  Value of c = {0}", c);

            c ^= 2;
            Console.WriteLine("Line 10 - ^=  Value of c = {0}", c);

            c |= 2;
            Console.WriteLine("Line 11 - |=  Value of c = {0}", c);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - =     Value of c = 21
Line 2 - +=    Value of c = 42
Line 3 - -=    Value of c = 21
Line 4 - *=    Value of c = 441
Line 5 - /=    Value of c = 21
Line 6 - %=    Value of c = 11
Line 7 - <<=   Value of c = 44
Line 8 - >>=   Value of c = 11
Line 9 - &=    Value of c = 2
Line 10 - ^=   Value of c = 0
Line 11 - |=   Value of c = 2
```

## Misc Operators

There are few other important operators including **sizeof, typeof** and **? :** supported by C#.

| Operator | Description | Example |
|---|---|---|
| sizeof() | Returns the size of a data type. | sizeof(int), will return 4. |
| typeof() | Returns the type of a class. | typeof(StreamReader); |
| & | Returns the address of an variable. | &a; will give actual address of the variable. |
| * | Pointer to a variable. | *a; will pointer to a variable. |
| ? : | Conditional Expression | If Condition is true ? Then value X : Otherwise value Y |
| is | Determines whether an object is of a certain type. | If( Ford is Car) // checks if Ford is an object of the Car class. |
| As | Cast without raising an exception if the cast fails. | Object obj = new StringReader("Hello");<br>StringReader r = obj as StringReader; |

### Example

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {

         /* example of sizeof operator */
         Console.WriteLine("The size of int is {0}", sizeof(int));
         Console.WriteLine("The size of short is {0}", sizeof(short));
         Console.WriteLine("The size of double is {0}", sizeof(double));

         /* example of ternary operator */
         int a, b;
         a = 10;
         b = (a == 1) ? 20 : 30;
         Console.WriteLine("Value of b is {0}", b);

         b = (a == 10) ? 20 : 30;
         Console.WriteLine("Value of b is {0}", b);
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
The size of int is 4
The size of short is 2
The size of double is 8
Value of b is 30
```

```
Value of b is 20
```

# Operators Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

| Category | Operator | Associativity |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* & sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %=>>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

## Example

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {
         int a = 20;
         int b = 10;
         int c = 15;
         int d = 5;
```

```
            int e;
            e = (a + b) * c / d;      // ( 30 * 15 ) / 5
            Console.WriteLine("Value of (a + b) * c / d is : {0}", e);

            e = ((a + b) * c) / d;    // (30 * 15 ) / 5
            Console.WriteLine("Value of ((a + b) * c) / d is  : {0}", e);

            e = (a + b) * (c / d);    // (30) * (15/5)
            Console.WriteLine("Value of (a + b) * (c / d) is  : {0}", e);

            e = a + (b * c) / d;     //  20 + (150/5)
            Console.WriteLine("Value of a + (b * c) / d is  : {0}", e);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50
```

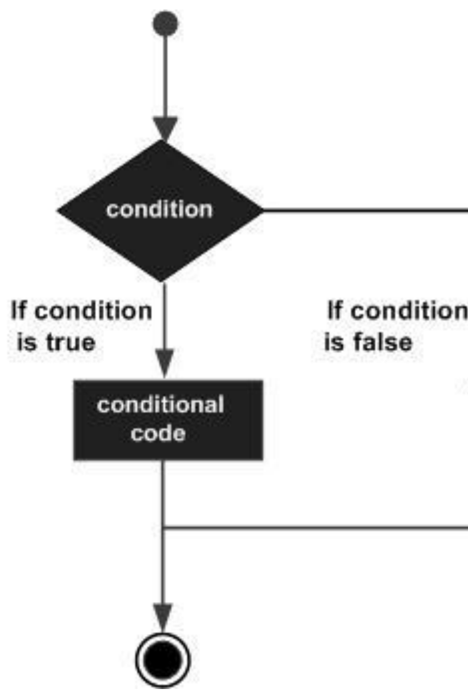
CHAPTER

10

# C# Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages:




C# provides following types of decision making statements. Click the following links to check their detail.

| Statement | Description |
|---|---|
| if statement | An **if statement** consists of a boolean expression followed by one or more statements. |
| if...else statement | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |

| | |
|---|---|
| nested if statements | You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| switch statement | A **switch** statement allows a variable to be tested for equality against a list of values. |
| nested switch statements | You can use one **swicth** statement inside another **switch** statement(s). |

# If statement

An **if** statement consists of a boolean expression followed by one or more statements.

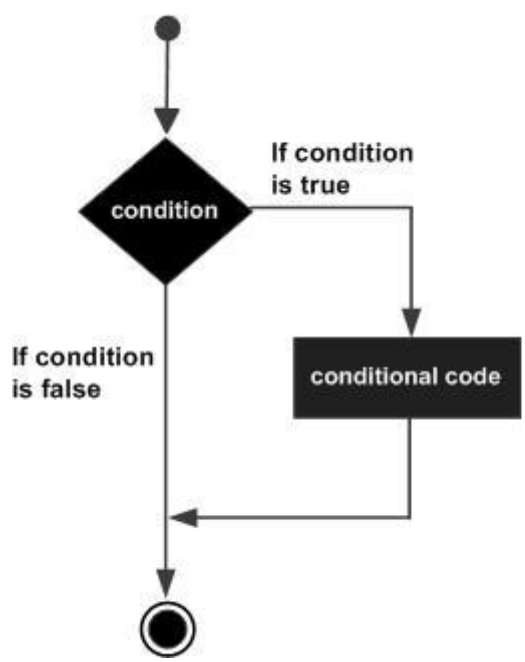
## Syntax:

The syntax of an if statement in C# is:

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
```

If the boolean expression evaluates to **true,** then the block of code inside the if statement will be executed. If boolean expression evaluates to **false** then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

## Flow Diagram:




## Example:

```
using System;

namespace DecisionMaking
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* check the boolean condition using if statement */
            if (a < 20)
            {
                /* if condition is true then print the following */
                Console.WriteLine("a is less than 20");
            }
            Console.WriteLine("value of a is : {0}", a);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20;
value of a is : 10
```

# If...else statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

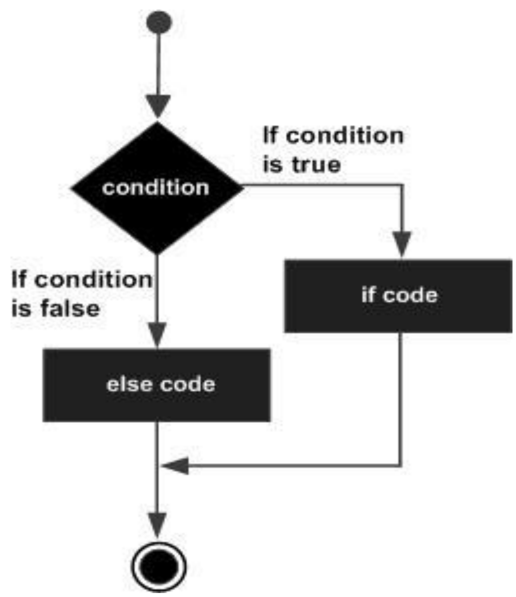
## Syntax:

The syntax of an **if...else** statement in C# is:

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
else
{
  /* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to **true,** then the **if block** of code will be executed otherwise **else block** of code will be executed.

## Flow Diagram:




## Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            /* local variable definition */
            int a = 100;

            /* check the boolean condition */
            if (a < 20)
            {
                /* if condition is true then print the following */
                Console.WriteLine("a is less than 20");
            }
            else
            {
                /* if condition is false then print the following */
                Console.WriteLine("a is not less than 20");
            }
            Console.WriteLine("value of a is : {0}", a);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;
value of a is : 100
```

## The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

## Syntax:

The syntax of an **if...else if...else** statement in C# is:

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```

## Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            /* local variable definition */
            int a = 100;

            /* check the boolean condition */
            if (a == 10)
            {
                /* if condition is true then print the following */
                Console.WriteLine("Value of a is 10");
            }
            else if (a == 20)
            {
                /* if else if condition is true */
```

```
                Console.WriteLine("Value of a is 20");
            }
            else if (a == 30)
            {
                /* if else if condition is true  */
                Console.WriteLine("Value of a is 30");
            }
            else
            {
                /* if none of the conditions is true */
                Console.WriteLine("None of the values is matching");
            }
            Console.WriteLine("Exact value of a is: {0}", a);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
None of the values is matching
Exact value of a is: 100
```

# nested if statements

It is always legal in C# to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

## Syntax:

The syntax for a **nested if** statement is as follows:

```
if( boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2)
   {
      /* Executes when the boolean expression 2 is true */
   }
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

## Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            //* local variable definition */
            int a = 100;
            int b = 200;
```

```
            /* check the boolean condition */
            if (a == 100)
            {
                /* if condition is true then check the following */
                if (b == 200)
                {
                    /* if condition is true then print the following */
                    Console.WriteLine("Value of a is 100 and b is 200");
                }
            }
            Console.WriteLine("Exact value of a is : {0}", a);
            Console.WriteLine("Exact value of b is : {0}", b);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

# switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

## Syntax:

The syntax for a **switch** statement in C# is as follows:

```
switch(expression){
    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
       statement(s);
}
```

The following rules apply to a **switch** statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

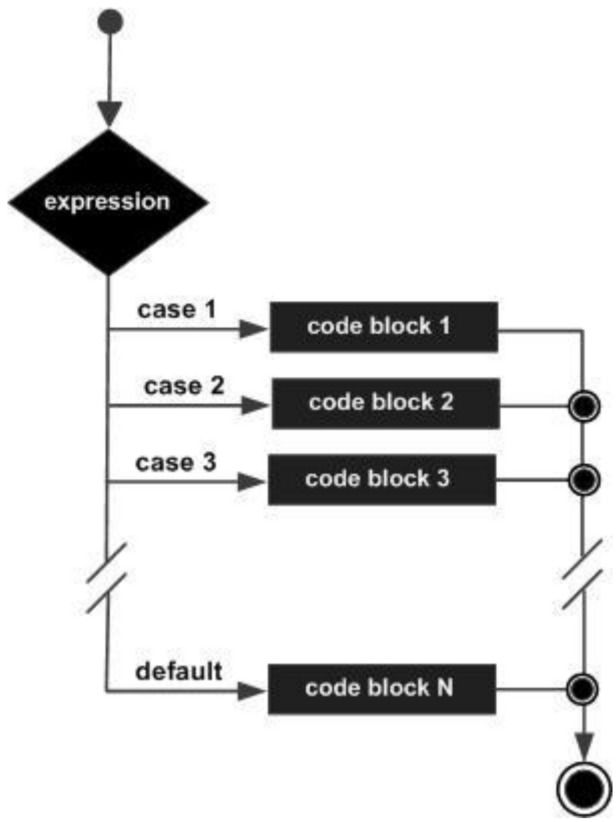
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through*to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

## Flow Diagram:




## Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            char grade = 'B';

            switch (grade)
            {
                case 'A':
                    Console.WriteLine("Excellent!");
                    break;
```

```
                case 'B':
                case 'C':
                    Console.WriteLine("Well done");
                    break;
                case 'D':
                    Console.WriteLine("You passed");
                    break;
                case 'F':
                    Console.WriteLine("Better try again");
                    break;
                default:
                    Console.WriteLine("Invalid grade");
                    break;
            }
            Console.WriteLine("Your grade is  {0}", grade);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Well done
Your grade is B
```

# nested switch statement

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

## Syntax:

The syntax for a **nested switch** statement is as follows:

```
switch(ch1)
{
   case 'A':
      printf("This A is part of outer switch" );
      switch(ch2)
      {
         case 'A':
            printf("This A is part of inner switch" );
            break;
         case 'B': /* inner B case code */
      }
      break;
   case 'B': /* outer B case code */
}
```

## Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
```

```
        static void Main(string[] args)
        {
            int a = 100;
            int b = 200;

            switch (a)
            {
                case 100:
                    Console.WriteLine("This is part of outer switch ");
                    switch (b)
                    {
                        case 200:
                        Console.WriteLine("This is part of inner switch ");
                        break;
                    }
                    break;
            }
            Console.WriteLine("Exact value of a is : {0}", a);
            Console.WriteLine("Exact value of b is : {0}", b);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

# The ? : Operator:

We have covered **conditional operator ? :** in previous chapter which can be used to replace **if...else**statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

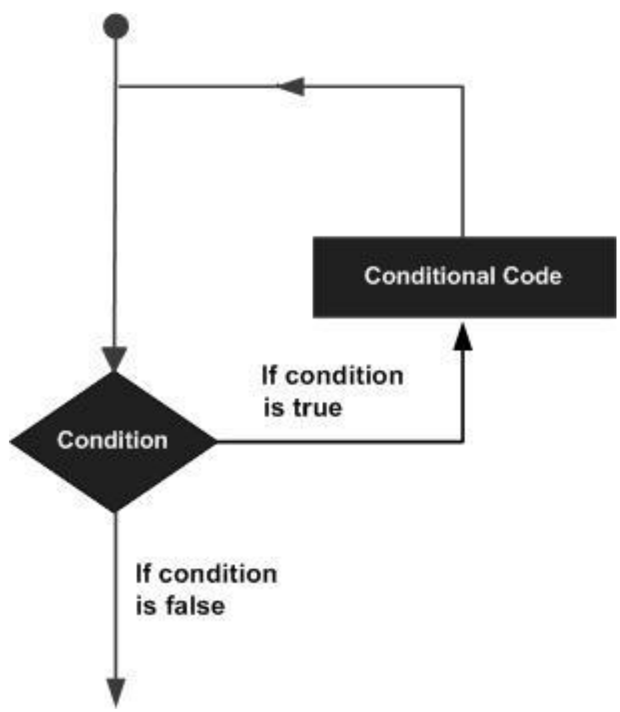
# CHAPTER 11

# C# Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:




C# provides following types of loop to handle looping requirements. Click the following links to check their detail.

| Loop Type | Description |
| --- | --- |

| | |
|---|---|
| while loop | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| for loop | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| do...while loop | Like a while statement, except that it tests the condition at the end of the loop body |
| nested loops | You can use one or more loop inside any another while, for or do..while loop. |

# while loop

A **while** loop statement in C# repeatedly executes a target statement as long as a given condition is true.

## Syntax:

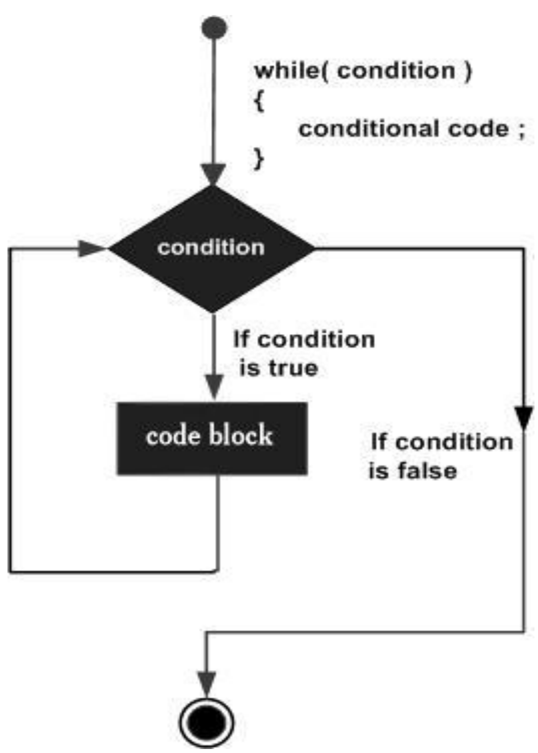
The syntax of a **while** loop in C# is:

```
while(condition)
{
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

## Flow Diagram:



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Example:

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* while loop execution */
            while (a < 20)
            {
                Console.WriteLine("value of a: {0}", a);
                a++;
            }
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

# for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

## Syntax:

The syntax of a **for** loop in C# is:

```
for ( init; condition; increment )
{
   statement(s);
}
```

Here is the flow of control in a for loop:

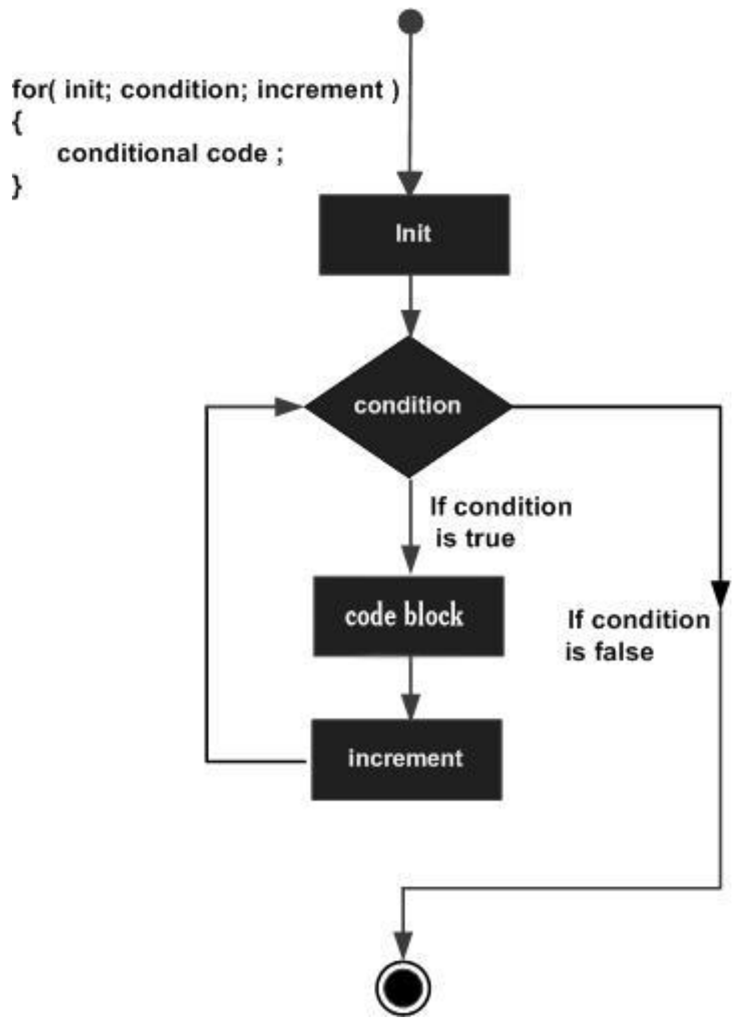
1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

## Flow Diagram:




## Example:

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* for loop execution */
```

```
            for (int a = 10; a < 20; a = a + 1)
            {
                Console.WriteLine("value of a: {0}", a);
            }
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

# do...while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

## Syntax:

The syntax of a **do...while** loop in C# is:

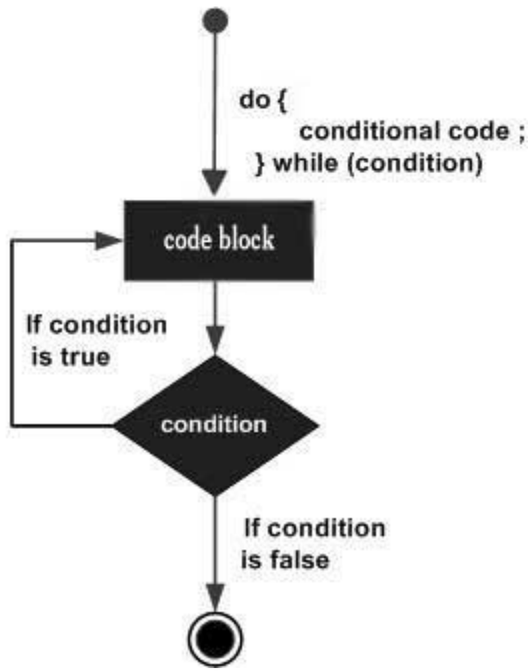
```
do
{
   statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

## Flow Diagram:




## Example:

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* do loop execution */
            do
            {
               Console.WriteLine("value of a: {0}", a);
                a = a + 1;
            } while (a < 20);

            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

```
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

# nested loops

C# allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

## Syntax:

The syntax for a **nested for loop** statement in C# is as follows:

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
   }
   statement(s);
}
```

The syntax for a **nested while loop** statement in C# is as follows:

```
while(condition)
{
   while(condition)
   {
      statement(s);
   }
   statement(s);
}
```

The syntax for a **nested do...while loop** statement in C# is as follows:

```
do
{
   statement(s);
   do
   {
      statement(s);
   }while( condition );

}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

## Example:

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
using System;

namespace Loops
{

   class Program
   {
```

```
        static void Main(string[] args)
        {
           /* local variable definition */
           int i, j;

           for (i = 2; i < 100; i++)
           {
              for (j = 2; j <= (i / j); j++)
                 if ((i % j) == 0) break; // if factor found, not prime
              if (j > (i / j))
                 Console.WriteLine("{0} is prime", i);
           }

           Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

## Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C# provides the following control statements. Click the following links to check their details.

| Control Statement | Description |
|---|---|
| break statement | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |

| | |
|---|---|
| continue statement | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

# break statement

The **break** statement in C# has following two usages:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement.

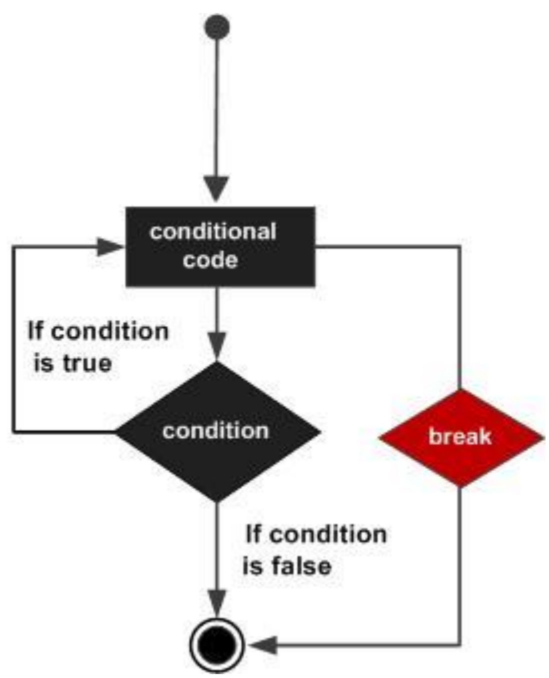
If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

## Syntax:

The syntax for a **break** statement in C# is as follows:

```
break;
```

## Flow Diagram:




## Example:

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;
```

```
            /* while loop execution */
            while (a < 20)
            {
                Console.WriteLine("value of a: {0}", a);
                a++;
                if (a > 15)
                {
                    /* terminate the loop using break statement */
                    break;
                }
            }
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

# continue statement

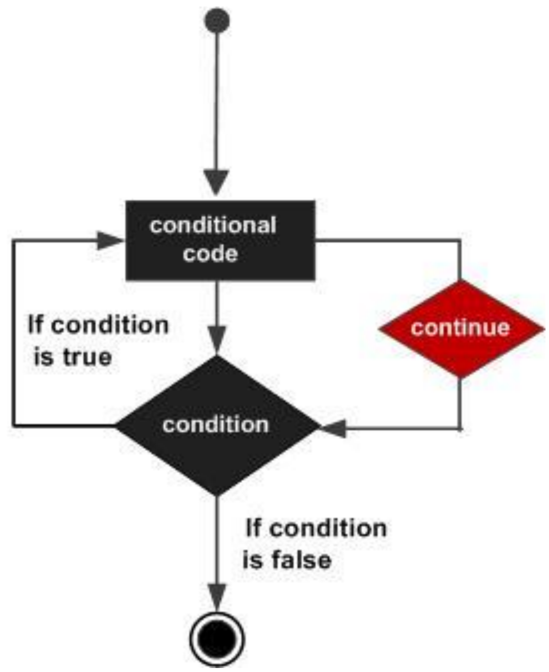
The **continue** statement in C# works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.
For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests.

## Syntax:

The syntax for a **continue** statement in C# is as follows:

```
continue;
```

## Flow Diagram:




## Example:

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 10;

            /* do loop execution */
            do
            {
                if (a == 15)
                {
                    /* skip the iteration */
                    a = a + 1;
                    continue;
                }
                Console.WriteLine("value of a: {0}", a);
                a++;

            } while (a < 20);

            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

# The Infinite Loop:

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Hey! I am Trapped");
            }

        }
    }
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.

CHAPTER

12

# C# Encapsulation

Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'.

Encapsulation, in object oriented programming methodology, prevents access to implementation details.

Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*. Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers:

- Public
- Private
- Protected
- Internal
- Protected internal

## Public Access Specifier

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.

The following example illustrates this:

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        public double length;
        public double width;

        public double GetArea()
        {
            return length * width;
```

```
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
                          r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In the preceding example, the member variables length and width are declared **public**, so they can be accessed from the function Main() using an instance of the Rectangle class, named **r**.

The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class.
The member functions *Display()* is also declared **public**, so it can also be accessed from *Main()* using an instance of the Rectangle class, named **r**.

# Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

The following example illustrates this:

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        private double length;
        private double width;

        public void Acceptdetails()
        {
            Console.WriteLine("Enter Length: ");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter Width: ");
            width = Convert.ToDouble(Console.ReadLine());
        }
```

```
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

In the preceding example, the member variables length and width are declared **private**, so they cannot be accessed from the function Main(). The member functions *AcceptDetails()* and *Display()* can access these variables. Since the member functions *AcceptDetails()* and *Display()* are declared **public**, they can be accessed from *Main()* using an instance of the Rectangle class, named **r**.

# Protected Access Specifier

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

# Internal Access Specifier

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

The following program illustrates this:

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        internal double length;
        internal double width;
```

```
        double GetArea()
        {
            return length * width;
        }
       public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In the preceding example, notice that the member function *GetArea()* is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is **private**.

## Protected Internal Access Specifier

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

CHAPTER

13

# C# Methods

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to:

- Define the method
- Call the method

## Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows:

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

Following are the various elements of a method:

- **Access Specifier**: This determines the visibility of a variable or a method from another class.
- **Return type**: A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
- **Method name**: Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list**: Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method body**: This contains the set of instructions needed to complete the required activity.

### Example:

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator
{
```

```
   public int FindMax(int num1, int num2)
   {
      /* local variable declaration */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

## Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int FindMax(int num1, int num2)
      {
         /* local variable declaration */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;

         return result;
      }
      static void Main(string[] args)
      {
         /* local variable definition */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();

         //calling the FindMax method
         ret = n.FindMax(a, b);
         Console.WriteLine("Max value is : {0}", ret );
         Console.ReadLine();
      }
   }
```

When the above code is compiled and executed, it produces the following result:

```
Max value is : 200
```

You can also call public method from other classes by using the instance of the class. For example, the method *FindMax* belongs to the *NumberManipulator* class, you can call it from another class *Test*.

```
using System;
```

```
namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int FindMax(int num1, int num2)
        {
            /* local variable declaration */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();
            //calling the FindMax method
            ret = n.FindMax(a, b);
            Console.WriteLine("Max value is : {0}", ret );
            Console.ReadLine();

        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Max value is : 200
```

# Recursive Method Call

A method can call itself. This is known as **recursion**. Following is an example that calculates factorial for a given number using a recursive function:

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public int factorial(int num)
        {
            /* local variable declaration */
            int result;

            if (num == 1)
            {
                return 1;
            }
```

```
            else
            {
                result = factorial(num - 1) * num;
                return result;
            }
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            //calling the factorial method
            Console.WriteLine("Factorial of 6 is : {0}", n.factorial(6));
            Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));
            Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));
            Console.ReadLine();

        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
```

# Passing Parameters to a Method

When method with parameters is called, you need to pass the parameters to the method. In C#, there are three ways that parameters can be passed to a method:

| Mechanism | Description |
|---|---|
| Value parameters | This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| Reference parameters | This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument. |
| Output parameters | TThis method helps in returning more than one value. |

# Value parameters

This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for each value parameter.

The values of the actual parameters are copied into them. So, the changes made to the parameter inside the method have no effect on the argument. The following example demonstrates the concept:

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public void swap(int x, int y)
```

```
        {
            int temp;

            temp = x; /* save the value of x */
            x = y;    /* put y into x */
            y = temp; /* put temp into y */
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            /* local variable definition */
            int a = 100;
            int b = 200;

            Console.WriteLine("Before swap, value of a : {0}", a);
            Console.WriteLine("Before swap, value of b : {0}", b);

            /* calling a function to swap the values */
            n.swap(a, b);

            Console.WriteLine("After swap, value of a : {0}", a);
            Console.WriteLine("After swap, value of b : {0}", b);

            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

# Reference parameters

A reference parameter is a **reference to a memory location** of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

In C#, you declare the reference parameters using the **ref** keyword. The following example demonstrates this:

```
using System;

namespace CalculatorApplication

{

   class NumberManipulator

   {

      public void swap(ref int x, ref int y)

      {

         int temp;
```

```
            temp = x; /* save the value of x */
            x = y;   /* put y into x */
            y = temp; /* put temp into y */
         }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* local variable definition */
         int a = 100;
         int b = 200;

         Console.WriteLine("Before swap, value of a : {0}", a);
         Console.WriteLine("Before swap, value of b : {0}", b);

         /* calling a function to swap the values */
         n.swap(ref a, ref b);

         Console.WriteLine("After swap, value of a : {0}", a);
         Console.WriteLine("After swap, value of b : {0}", b);

         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

It shows that the values have been changed inside the *swap* function and this change reflects in the *Main* function.

## Output parameters

A return statement can be used for returning only one value from a function. However, using **output parameters**, you can return two values from a function. Output parameters are like reference parameters, except that they transfer data out of the method rather than into it.

The following example illustrates this:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValue(out int x )
      {
         int temp = 5;
         x = temp;
      }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* local variable definition */
         int a = 100;

         Console.WriteLine("Before method call, value of a : {0}", a);

         /* calling a function to get the value */
         n.getValue(out a);

         Console.WriteLine("After method call, value of a : {0}", a);
         Console.ReadLine();

      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Before method call, value of a : 100
After method call, value of a : 5
```

The variable supplied for the output parameter need not be assigned a value the method call. Output parameters are particularly useful when you need to return values from a method through the parameters without assigning an initial value to the parameter. Look at the following example, to understand this:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValues(out int x, out int y )
      {
          Console.WriteLine("Enter the first value: ");
          x = Convert.ToInt32(Console.ReadLine());
          Console.WriteLine("Enter the second value: ");
          y = Convert.ToInt32(Console.ReadLine());
      }
```

```
        static void Main(string[] args)
        {
           NumberManipulator n = new NumberManipulator();
           /* local variable definition */
           int a , b;

           /* calling a function to get the values */
           n.getValues(out a, out b);

           Console.WriteLine("After method call, value of a : {0}", a);
           Console.WriteLine("After method call, value of b : {0}", b);
           Console.ReadLine();
        }
     }
  }
```

When the above code is compiled and executed, it produces the following result (depending upon the user input):

```
Enter the first value:
7
Enter the second value:
8
After method call, value of a : 7
After method call, value of b : 8
```

CHAPTER

14

# C# Nullables

C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.

For example, you can store any value from -2,147,483,648 to 2,147,483,647 or null in a Nullable< Int32 > variable. Similarly, you can assign true, false or null in a Nullable< bool > variable. Syntax for declaring a **nullable** type is as follows:

```
< data_type> ? <variable_name> = null;
```

The following example demonstrates use of nullable data types:

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {
      static void Main(string[] args)
      {
         int? num1 = null;
         int? num2 = 45;
         double? num3 = new double?();
         double? num4 = 3.14157;

         bool? boolval = new bool?();

         // display the values

         Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}",
                              num1, num2, num3, num4);
         Console.WriteLine("A Nullable boolean value: {0}", boolval);
         Console.ReadLine();

      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Nullables at Show: , 45,  , 3.14157
A Nullable boolean value:
```

## The Null Coalescing Operator (??)

The null coalescing operator is used with the nullable value types and reference types. It is used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.

If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand. The following example explains this:

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {

      static void Main(string[] args)
      {

         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;
         Console.WriteLine(" Value of num3: {0}", num3);
         num3 = num2 ?? 5.34;
         Console.WriteLine(" Value of num3: {0}", num3);
         Console.ReadLine();

      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of num3: 5.34
Value of num3: 3.14157
```

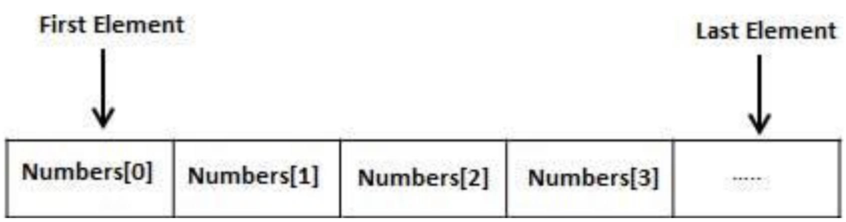
CHAPTER 15

# C# Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.




## Declaring Arrays

To declare an array in C#, you can use the following syntax:

```
datatype[] arrayName;
```

where,

- d*atatype* is used to specify the type of elements to be stored in the array.
- *[ ]* specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

```
double[] balance;
```

## Initializing an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array.

Array is a reference type, so you need to use the **new** keyword to create an instance of the array.

For example,

```
double[] balance = new double[10];
```

## Assigning Values to an Array

You can assign values to individual array elements, by using the index number, like:

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

You can assign values to the array at the time of declaration, like:

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

You can also create and initialize an array, like:

```
int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

In the preceding case, you may also omit the size of the array, like:

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

You can also copy an array variable into another target array variable. In that case, both the target and source would point to the same memory location:

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

When you create an array, C# compiler implicitly initializes each array element to a default value depending on the array type. For example for an int array all elements would be initialized to 0.

## Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n is an array of 10 integers */
         int i,j;


         /* initialize elements of array n */
         for ( i = 0; i < 10; i++ )
         {
            n[ i ] = i + 100;
         }
```

```
            /* output each array element's value */
            for (j = 0; j < 10; j++ )
            {
               Console.WriteLine("Element[{0}] = {1}", j, n[j]);
            }
            Console.ReadKey();
         }
      }
   }
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

# Using the *foreach* Loop

In the previous example, we have used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n is an array of 10 integers */


         /* initialize elements of array n */
         for ( int i = 0; i < 10; i++ )
         {
            n[i] = i + 100;
         }

         /* output each array element's value */
         foreach (int j in n )
         {
            int i = j-100;
            Console.WriteLine("Element[{0}] = {1}", i, j);
            i++;
         }
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

# C# Arrays in Detail

Arrays are important to C# and should need lots of more detail. There are following few important concepts related to array which should be clear to a C# programmer:

| Concept | Description |
|---|---|
| Multi-dimensional arrays | C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| Jagged arrays | C# supports multidimensional arrays, which are arrays of arrays. |
| Passing arrays to functions | You can pass to the function a pointer to an array by specifying the array's name without an index. |
| Param arrays | This is used for passing unknown number of parameters to a function. |
| The Array Class | Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays. |

# Multi-dimensional arrays

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array.

You can declare a 2-dimensional array of strings as:

```
string [,] names;
```

or, a 3-dimensional array of int variables:

```
int [ , , ] m;
```

## Two-Dimensional Arrays:

The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is, in essence, a list of one-dimensional arrays.

A 2-dimensional array can be thought of as a table, which will have x number of rows and y number of columns. Following is a 2-dimentional array, which contains 3 rows and 4 columns:

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

Thus, every element in array a is identified by an element name of the form a[ i , j ], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

## Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int [,] a = int [3,4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

## Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2,3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
using System;

namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            /* an array with 5 rows and 2 columns*/
            int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };

            int i, j;

            /* output each array element's value */
            for (i = 0; i < 5; i++)
            {
                for (j = 0; j < 2; j++)
                {
                    Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);
                }
            }
           Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0,0]: 0
a[0,1]: 0
a[1,0]: 1
a[1,1]: 2
a[2,0]: 2
a[2,1]: 4
a[3,0]: 3
a[3,1]: 6
a[4,0]: 4
a[4,1]: 8
```

# Jagged Arrays

A Jagged array is an array of arrays. You can declare a jagged array *scores* of **int** values as:

```
int [][] scores;
```

Declaring an array, does not create the array in memory. To create the above array:

```
int[][] scores = new int[5][];
for (int i = 0; i < scores.Length; x++)
{
   scores[i] = new int[4];
}
```

You can initialize a jagged array as:

```
int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};
```

Where, scores is an array of two arrays of integers -- scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

## Example

The following example illustrates using a jagged array:

```
using System;

namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            /* a jagged array of 5 array of integers*/
            int[][] a = new int[][]{new int[]{0,0},new int[]{1,2},
            new int[]{2,4},new int[]{ 3, 6 }, new int[]{ 4, 8 } };

            int i, j;

            /* output each array element's value */
            for (i = 0; i < 5; i++)
            {
                for (j = 0; j <; 2; j++)
                {
                    Console.WriteLine("a[{0}][{1}] = {2}", i, j, a[i][j]);
                }
            }
```

```
            Console.ReadKey();
         }
      }
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

## Passing arrays to functions

You can pass an array as a function argument in C#. The following example demonstrates this:

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
      double getAverage(int[] arr, int size)
      {
         int i;
         double avg;
         int sum = 0;

         for (i = 0; i < size; ++i)
         {
            sum += arr[i];
         }

         avg = (double)sum / size;
         return avg;
      }
      static void Main(string[] args)
      {
         MyArray app = new MyArray();
         /* an int array with 5 elements */
         int [] balance = new int[]{1000, 2, 3, 17, 50};
         double avg;

         /* pass pointer to the array as an argument */
         avg = app.getAverage(balance, 5 ) ;

         /* output the returned value */
         Console.WriteLine( "Average value is: {0} ", avg );
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Average value is: 214.4
```

# Param arrays

At times, while declaring a method, you are not sure of the number of arguments passed as a parameter. C# param arrays (or parameter arrays) come into help at these times.

The following example demonstrates this:

```
using System;

namespace ArrayApplication
{
   class ParamArray
   {
      public int AddElements(params int[] arr)
      {
         int sum = 0;
         foreach (int i in arr)
         {
            sum += i;
         }
         return sum;
      }
   }

   class TestClass
   {
      static void Main(string[] args)
      {
         ParamArray app = new ParamArray();
         int sum = app.AddElements(512, 720, 250, 567, 889);
         Console.WriteLine("The sum is: {0}", sum);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
The sum is: 2938
```

# The Array Class

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

## Properties of the Array Class

The following table provides some of the most commonly used properties of the Array class:

| S.N | Property Name & Description |
|---|---|
| 1 | **IsFixedSize**<br>Gets a value indicating whether the Array has a fixed size. |

| | |
|---|---|
| 2 | **IsReadOnly**<br>Gets a value indicating whether the Array is read-only. |
| 3 | **Length**<br>Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array. |
| 4 | **LongLength**<br>Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array. |
| 5 | **Rank**<br>Gets the rank (number of dimensions) of the Array. |

## Methods of the Array Class

The following table provides some of the most commonly used properties of the Array class:

| S.N | Method Name & Description |
|---|---|
| 1 | **Clear**<br>Sets a range of elements in the Array to zero, to false, or to null, depending on the element type. |
| 2 | **Copy(Array, Array, Int32)**<br>Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer. |
| 3 | **CopyTo(Array, Int32)**<br>Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer. |
| 4 | **GetLength**<br>Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array. |
| 5 | **GetLongLength**<br>Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array. |
| 6 | **GetLowerBound**<br>Gets the lower bound of the specified dimension in the Array. |
| 7 | **GetType**<br>Gets the Type of the current instance. (Inherited from Object.) |
| 8 | **GetUpperBound**<br>Gets the upper bound of the specified dimension in the Array. |
| 9 | **GetValue(Int32)**<br>Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer. |
| 10 | **IndexOf(Array, Object)**<br>Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array. |
| 11 | **Reverse(Array)**<br>Reverses the sequence of the elements in the entire one-dimensional Array. |
| 12 | **SetValue(Object, Int32)**<br>Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer. |

| | |
|---|---|
| 13 | **Sort(Array)**<br>Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array. |
| 14 | **ToStringk**<br>Returns a string that represents the current object. (Inherited from Object.) |

For complete list of Array class properties and methods, please consult Microsoft documentation on C#.

## Example

The following program demonstrates use of some of the methods of the Array class:

```
using System;
namespace ArrayApplication
{
    class MyArray
    {

        static void Main(string[] args)
        {
            int[] list = { 34, 72, 13, 44, 25, 30, 10 };
            int[] temp = list;

            Console.Write("Original Array: ");
            foreach (int i in list)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();

            // reverse the array
            Array.Reverse(temp);
            Console.Write("Reversed Array: ");
            foreach (int i in temp)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();

            //sort the array
            Array.Sort(list);
            Console.Write("Sorted Array: ");
            foreach (int i in list)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();

           Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Original Array: 34 72 13 44 25 30 10
Reversed Array: 10 30 25 44 13 72 34
Sorted Array: 10 13 25 30 34 44 72
```

CHAPTER

16

# C# Strings

In C#, you can use strings as array of characters, however, more common practice is to use the **string** keyword to declare a string variable. The string keyword is an alias for the **System.String** class.

## Creating a String Object

You can create string object using one of the following methods:

- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By retrieving a property or calling a method that returns a string
- By calling a formatting method to convert a value or object to its string representation

The following example demonstrates this:

```
using System;

namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //from string literal and string concatenation
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //by using string constructor
            char[] letters = { 'H', 'e', 'l', 'l','o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);
```

```
            //methods returning string
            string[] sarray = { "Hello", "From", "Tutorials", "Point" };
            string message = String.Join(" ", sarray);
            Console.WriteLine("Message: {0}", message);

            //formatting method to convert a value
            DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
            string chat = String.Format("Message sent at {0:t} on {0:D}",
            waiting);
            Console.WriteLine("Message: {0}", chat);
            Console.ReadKey() ;
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012
```

## Properties of the String Class

The String class has the following two properties:

| S.N | Property Name & Description |
|---|---|
| 1 | **Chars**<br>Gets the *Char* object at a specified position in the current *String* object. |
| 2 | **Length**<br>Gets the number of characters in the current String object. |

## Methods of the String Class

The String class has numerous methods that help you in working with the string objects. The following table provides some of the most commonly used methods:

| S.N | Method Name & Description |
|---|---|
| 1 | **public static int Compare( string strA, string strB )**<br>Compares two specified string objects and returns an integer that indicates their relative position in the sort order. |
| 2 | **public static int Compare( string strA, string strB, bool ignoreCase )**<br>Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true. |
| 3 | **public static string Concat( string str0, string str1 )**<br>Concatenates two string objects. |
| 4 | **public static string Concat( string str0, string str1, string str2 )**<br>Concatenates three string objects. |
| 5 | **public static string Concat( string str0, string str1, string str2, string str3 )**<br>Concatenates four string objects. |

| | |
|---|---|
| 6 | **public bool Contains( string value )**<br>Returns a value indicating whether the specified string object occurs within this string. |
| 7 | **public static string Copy( string str )**<br>Creates a new String object with the same value as the specified string. |
| 8 | **public void CopyTo( int sourceIndex, char[] destination, int destinationIndex, int count )**<br>Copies a specified number of characters from a specified position of the string object to a specified position in an array of Unicode characters. |
| 9 | **public bool EndsWith( string value )**<br>Determines whether the end of the string object matches the specified string. |
| 10 | **public bool Equals( string value )**<br>Determines whether the current string object and the specified string object have the same value. |
| 11 | **public static bool Equals( string a, string b )**<br>Determines whether two specified string objects have the same value. |
| 12 | **public static string Format( string format, Object arg0 )**<br>Replaces one or more format items in a specified string with the string representation of a specified object. |
| 13 | **public int IndexOf( char value )**<br>Returns the zero-based index of the first occurrence of the specified Unicode character in the current string. |
| 14 | **public int IndexOf( string value )**<br>Returns the zero-based index of the first occurrence of the specified string in this instance. |
| 15 | **public int IndexOf( char value, int startIndex )**<br>Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position. |
| 16 | **public int IndexOf( string value, int startIndex )**<br>Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position. |
| 17 | **public int IndexOfAny( char[] anyOf )**<br>Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters. |
| 18 | **public int IndexOfAny( char[] anyOf, int startIndex )**<br>Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position. |
| 19 | **public string Insert( int startIndex, string value )**<br>Returns a new string in which a specified string is inserted at a specified index position in the current string object. |
| 20 | **public static bool IsNullOrEmpty( string value )**<br>Indicates whether the specified string is null or an Empty string. |
| 21 | **public static string Join( string separator, params string[] value )**<br>Concatenates all the elements of a string array, using the specified separator between each element. |
| 22 | **public static string Join( string separator, string[] value, int startIndex, int count )**<br>Concatenates the specified elements of a string array, using the specified separator between each element. |
| 23 | **public int LastIndexOf( char value )**<br>Returns the zero-based index position of the last occurrence of the specified Unicode character within the current string object. |

| | |
|---|---|
| 24 | **public int LastIndexOf( string value )**<br>Returns the zero-based index position of the last occurrence of a specified string within the current string object. |
| 25 | **public string Remove( int startIndex )**<br>Removes all the characters in the current instance, beginning at a specified position and continuing through the last position, and returns the string. |
| 26 | **public string Remove( int startIndex, int count )**<br>Removes the specified number of characters in the current string beginning at a specified position and returns the string. |
| 27 | **public string Replace( char oldChar, char newChar )**<br>Replaces all occurrences of a specified Unicode character in the current string object with the specified Unicode character and returns the new string. |
| 28 | **public string Replace( string oldValue, string newValue )**<br>Replaces all occurrences of a specified string in the current string object with the specified string and returns the new string. |
| 29 | **public string[] Split( params char[] separator )**<br>Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. |
| 30 | **public string[] Split( char[] separator, int count )**<br>Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. The int parameter specifies the maximum number of substrings to return. |
| 31 | **public bool StartsWith( string value )**<br>Determines whether the beginning of this string instance matches the specified string. |
| 32 | **public char[] ToCharArray()**<br>Returns a Unicode character array with all the characters in the current string object. |
| 33 | **public char[] ToCharArray( int startIndex, int length )**<br>Returns a Unicode character array with all the characters in the current string object, starting from the specified index and up to the specified length. |
| 34 | **public string ToLower()**<br>Returns a copy of this string converted to lowercase. |
| 35 | **public string ToUpper()**<br>Returns a copy of this string converted to uppercase. |
| 36 | **public string Trim()**<br>Removes all leading and trailing white-space characters from the current String object. |

The above list of methods is not exhaustive, please visit MSDN library for the complete list of methods and String class constructors.

# Examples:

The following example demonstrates some of the methods mentioned above:

## Comparing Strings:

```
using System;

namespace StringApplication
{
   class StringProg
```

```
    {
        static void Main(string[] args)
        {
            string str1 = "This is test";
            string str2 = "This is text";

            if (String.Compare(str1, str2) == 0)
            {
                Console.WriteLine(str1 + " and " + str2 +  " are equal.");
            }
            else
            {
                Console.WriteLine(str1 + " and " + str2 + " are not equal.");
            }
            Console.ReadKey() ;
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
This is test and This is text are not equal.
```

## String Contains String:

```
using System;

namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "This is test";
            if (str.Contains("test"))
            {
                Console.WriteLine("The sequence 'test' was found.");
            }
            Console.ReadKey() ;
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
The sequence 'test' was found.
```

## Getting a Substring:

```
using System;

namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "Last night I dreamt of San Pedro";
            Console.WriteLine(str);
```

```
            string substr = str.Substring(23);
            Console.WriteLine(substr);
         }
         Console.ReadKey() ;
      }
   }
```

When the above code is compiled and executed, it produces the following result:

```
San Pedro
```

## Joining Strings:

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark",
         "And the sun shines daily on the mountain top",
         "I took a trip on a sailing ship",
         "And when I reached Jamaica",
         "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
      }
      Console.ReadKey() ;
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

CHAPTER

17

# C# Structures

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

## Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you would declare the Book structure:

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

The following program shows the use of the structure:

```
using System;

struct Books
{
   public string title;
   public string author;
   public string subject;
```

```
   public int book_id;
};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1;        /* Declare Book1 of type Book */
      Books Book2;        /* Declare Book2 of type Book */

      /* book 1 specification */
      Book1.title = "C Programming";
      Book1.author = "Nuha Ali";
      Book1.subject = "C Programming Tutorial";
      Book1.book_id = 6495407;

      /* book 2 specification */
      Book2.title = "Telecom Billing";
      Book2.author = "Zara Ali";
      Book2.subject =  "Telecom Billing Tutorial";
      Book2.book_id = 6495700;

      /* print Book1 info */
      Console.WriteLine( "Book 1 title : {0}", Book1.title);
      Console.WriteLine("Book 1 author : {0}", Book1.author);
      Console.WriteLine("Book 1 subject : {0}", Book1.subject);
      Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

      /* print Book2 info */
      Console.WriteLine("Book 2 title : {0}", Book2.title);
      Console.WriteLine("Book 2 author : {0}", Book2.author);
      Console.WriteLine("Book 2 subject : {0}", Book2.subject);
      Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

      Console.ReadKey();

   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

# Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features:

- Structures can have methods, fields, indexers, properties, operator methods, and events.

- Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and can't be changed.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.
- Structure members cannot be specified as abstract, virtual, or protected.
- When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
- If the New operator is not used, the fields will remain unassigned and the object cannot be used until all the fields are initialized.

# Class vs Structure

Classes and Structures have the following basic differences:

- classes are reference types and structs are value types
- structures do not support inheritance
- structures cannot have default constructor

In the light of the above discussions, let us rewrite the previous example:

```
using System;

struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;
   public void getValues(string t, string a, string s, int id)
   {
      title = t;
      author = a;
      subject = s;
      book_id = id;
   }
   public void display()
   {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }

};

public class testStructure
{
```

```
    public static void Main(string[] args)
    {

       Books Book1 = new Books(); /* Declare Book1 of type Book */
       Books Book2 = new Books(); /* Declare Book2 of type Book */

       /* book 1 specification */
       Book1.getValues("C Programming",
       "Nuha Ali", "C Programming Tutorial",6495407);

       /* book 2 specification */
       Book2.getValues("Telecom Billing",
       "Zara Ali", "Telecom Billing Tutorial", 6495700);

       /* print Book1 info */
       Book1.display();

       /* print Book2 info */
       Book2.display();

       Console.ReadKey();

    }
 }
```

When the above code is compiled and executed, it produces the following result:

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

CHAPTER

18

# C# Enums

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

## Declaring *enum* Variable

The general syntax for declaring an enumeration is:

```
enum <enum_name>
{
    enumeration list
};
```

Where,

- The *enum_name* specifies the enumeration type name.
- The *enumeration list* is a comma-separated list of identifiers.

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. For example:

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

### Example:

The following example demonstrates use of enum variable:

```
using System;
namespace EnumApplication
{
   class EnumProgram
   {
      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

      static void Main(string[] args)
      {
         int WeekdayStart = (int)Days.Mon;
         int WeekdayEnd = (int)Days.Fri;
         Console.WriteLine("Monday: {0}", WeekdayStart);
```

```
            Console.WriteLine("Friday: {0}", WeekdayEnd);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Monday: 1
Friday: 5
```

CHAPTER

19

# C# Classes

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

## Class Definition

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. Following is the general form of a class definition:

```
<access specifier> class  class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    <access specifier> <data type> variableN;
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
    <access specifier> <return type> methodN(parameter_list)
    {
        // method body
    }
}
```

Please note that,

- Access specifiers specify the access rules for the members as well as the class itself, if not mentioned then the default access specifier for a class type is **internal**. Default access for the members is **private**.

- Data type specifies the type of variable, and return type specifies the data type of the data, the method returns, if any.

- To access the class members, you will use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far:

```
using System;
namespace BoxApplication
{
    class Box
    {
       public double length;   // Length of a box
       public double breadth;  // Breadth of a box
       public double height;   // Height of a box
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // Declare Box1 of type Box
            Box Box2 = new Box();        // Declare Box2 of type Box
            double volume = 0.0;         // Store the volume of a box here

            // box 1 specification
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // box 2 specification
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // volume of box 1
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Volume of Box1 : {0}",  volume);

            // volume of box 2
            volume = Box2.height * Box2.length * Box2.breadth;
            Console.WriteLine("Volume of Box2 : {0}", volume);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

# Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class:

```
using System;
namespace BoxApplication
{
    class Box
    {
       private double length;   // Length of a box
       private double breadth;  // Breadth of a box
       private double height;   // Height of a box
       public void setLength( double len )
       {
            length = len;
       }

       public void setBreadth( double bre )
       {
            breadth = bre;
       }

       public void setHeight( double hei )
       {
            height = hei;
       }
       public double getVolume()
       {
           return length * breadth * height;
       }
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // Declare Box1 of type Box
            Box Box2 = new Box();
            double volume;


            // Declare Box2 of type Box
            // box 1 specification
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // box 2 specification
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

            // volume of box 1
            volume = Box1.getVolume();
            Console.WriteLine("Volume of Box1 : {0}" ,volume);

            // volume of box 2
            volume = Box2.getVolume();
            Console.WriteLine("Volume of Box2 : {0}", volume);

            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

# Constructors in C#

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type. Following example explains the concept of constructor:

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()
      {
         Console.WriteLine("Object is being created");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
```

A **default constructor** does not have any parameter but if you need a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line(double len)  //Parameterized constructor
```

```
      {
         Console.WriteLine("Object is being created, length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

# Destructors in C#

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor:

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // Length of a line
      public Line()  // constructor
      {
         Console.WriteLine("Object is being created");
      }
      ~Line() //destructor
      {
         Console.WriteLine("Object is being deleted");
      }

      public void setLength( double len )
      {
```

```
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

# Static Members of a C# Class

We can define class members as static using the **static** keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.
The keyword **static** implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

The following example demonstrates the use of **static variables**:

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s1 = new StaticVar();
            StaticVar s2 = new StaticVar();
            s1.count();
            s1.count();
            s1.count();
            s2.count();
            s2.count();
```

```
            s2.count();
            Console.WriteLine("Variable num for s1: {0}", s1.getNum());
            Console.WriteLine("Variable num for s2: {0}", s2.getNum());
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Variable num for s1: 6
Variable num for s2: 6
```

You can also declare a **member function** as **static**. Such functions can access only static variables. The static functions exist even before the object is created. The following example demonstrates the use of **static functions**:

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
       public static int num;
        public void count()
        {
            num++;
        }
        public static int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s = new StaticVar();
            s.count();
            s.count();
            s.count();
            Console.WriteLine("Variable num: {0}", StaticVar.getNum());
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Variable num: 3
```

CHAPTER

20

# C# Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A**mammal hence dog **IS-A** animal as well and so on.

## Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base class or interface.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
 ...
}
class <derived_class> : <base_class>
{
 ...
}
```

Consider a base class Shape and its derived class Rectangle:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
```

```
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Derived class
   class Rectangle: Shape
   {
      public int getArea()
      {
         return (width * height);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```

# Base Class Initialization

The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created. You can give instructions for superclass initialization in the member initialization list.

The following program demonstrates this:

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      //member variables
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      public double GetArea()
      {
         return length * width;
```

```
      }
      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle
   class Tabletop : Rectangle
   {
      private double cost;
      public Tabletop(double l, double w) : base(l, w)
      { }
      public double GetCost()
      {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }
      public void Display()
      {
         base.Display();
         Console.WriteLine("Cost: {0}", GetCost());
      }
   }
   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Tabletop t = new Tabletop(4.5, 7.5);
         t.Display();
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

# Multiple Inheritance in C#

**C# does not support multiple inheritance**. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
         height = h;
```

```
      }
      protected int width;
      protected int height;
   }

   // Base class PaintCost
   public interface PaintCost
   {
      int getCost(int area);

   }
   // Derived class
   class Rectangle : Shape, PaintCost
   {
      public int getArea()
      {
         return (width * height);
      }
      public int getCost(int area)
      {
         return area * 70;
      }
   }
   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle Rect = new Rectangle();
         int area;
         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();
         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
Total paint cost: $2450
```

CHAPTER

21

# C# Polymorphism

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.
Polymorphism can be static or dynamic. In **static polymorphism,** the response to a function is determined at the compile time. In **dynamic polymorphism,** it is decided at run-time.

## Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. These are:

- Function overloading
- Operator overloading

We will discuss function overloading in the next section and operator overloading will be dealt with in next chapter.

## Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

```
using System;
namespace PolymorphismApplication
{
   class Printdata
   {
      void print(int i)
      {
         Console.WriteLine("Printing int: {0}", i );
      }

      void print(double f)
      {
         Console.WriteLine("Printing float: {0}" , f);
      }

      void print(string s)
      {
```

```
            Console.WriteLine("Printing string: {0}", s);
         }
         static void Main(string[] args)
         {
            Printdata p = new Printdata();
            // Call print to print integer
            p.print(5);
            // Call print to print float
            p.print(500.263);
            // Call print to print string
            p.print("Hello C++");
            Console.ReadKey();
         }
      }
   }
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

# Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Please note the following rules about abstract classes:

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class:

```
using System;
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
         Console.WriteLine("Rectangle class area :");
         return (width * length);
      }
   }
```

```
   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle(10, 7);
         double a = r.area();
         Console.WriteLine("Area: {0}",a);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area :
Area: 70
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this:

```
using System;
namespace PolymorphismApplication
{
   class Shape
   {
      protected int width, height;
      public Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      public virtual int area()
      {
         Console.WriteLine("Parent class area :");
         return 0;
      }
   }
   class Rectangle: Shape
   {
      public Rectangle( int a=0, int b=0): base(a, b)
      {

      }
      public override int area ()
      {
         Console.WriteLine("Rectangle class area :");
         return (width * height);
      }
   }
   class Triangle: Shape
   {
      public Triangle(int a = 0, int b = 0): base(a, b)
      {

      }
```

```
      public override int area()
      {
         Console.WriteLine("Triangle class area :");
         return (width * height / 2);
      }
   }
   class Caller
   {
      public void CallArea(Shape sh)
      {
         int a;
         a = sh.area();
         Console.WriteLine("Area: {0}", a);
      }
   }
   class Tester
   {

      static void Main(string[] args)
      {
         Caller c = new Caller();
         Rectangle r = new Rectangle(10, 7);
         Triangle t = new Triangle(10, 5);
         c.CallArea(r);
         c.CallArea(t);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

CHAPTER

22

# C# Operator Overloading

You can redefine or overload most of the built-in operators available in C#. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

For example, look at the following function:

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

The above function implements the addition operator (+) for a user-defined class Box. It adds the attributes of two Box objects and returns the resultant Box object.

## Implementation of Operator Overloading

The following program shows the complete implementation:

```
using System;

namespace OperatorOvlApplication
{
   class Box
   {
      private double length;      // Length of a box
      private double breadth;     // Breadth of a box
      private double height;      // Height of a box

      public double getVolume()
      {
         return length * breadth * height;
      }
      public void setLength( double len )
      {
         length = len;
      }
```

```
      public void setBreadth( double bre )
      {
         breadth = bre;
      }

      public void setHeight( double hei )
      {
         height = hei;
      }
      // Overload + operator to add two Box objects.
      public static Box operator+ (Box b, Box c)
      {
         Box box = new Box();
         box.length = b.length + c.length;
         box.breadth = b.breadth + c.breadth;
         box.height = b.height + c.height;
         return box;
      }

   }

   class Tester
   {
      static void Main(string[] args)
      {
         Box Box1 = new Box();         // Declare Box1 of type Box
         Box Box2 = new Box();         // Declare Box2 of type Box
         Box Box3 = new Box();         // Declare Box3 of type Box
         double volume = 0.0;          // Store the volume of a box here

         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}", volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         // Add two object as follows:
         Box3 = Box1 + Box2;

         // volume of box 3
         volume = Box3.getVolume();
         Console.WriteLine("Volume of Box3 : {0}", volume);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

# Overloadable and Non-Overloadable Operators

The following table describes the overload ability of the operators in C#:

| Operators | Description |
|---|---|
| +, -, !, ~, ++, -- | These unary operators take one operand and can be overloaded. |
| +, -, *, /, % | These binary operators take one operand and can be overloaded. |
| ==, !=, <, >, <=, >= | The comparison operators can be overloaded |
| &&, \|\| | The conditional logical operators cannot be overloaded directly. |
| +=, -=, *=, /=, %= | The assignment operators cannot be overloaded. |
| =, ., ?:, ->, new, is, sizeof, typeof | These operators cannot be overloaded. |

## Example:

In the light of the above discussions, let us extend the preceding example, and overload few more operators:

```
using System;

namespace OperatorOvlApplication
{
    class Box
    {
       private double length;      // Length of a box
       private double breadth;     // Breadth of a box
       private double height;      // Height of a box

       public double getVolume()
       {
         return length * breadth * height;
       }
      public void setLength( double len )
      {
          length = len;
      }

      public void setBreadth( double bre )
      {
          breadth = bre;
      }

      public void setHeight( double hei )
      {
          height = hei;
      }
      // Overload + operator to add two Box objects.
      public static Box operator+ (Box b, Box c)
      {
          Box box = new Box();
          box.length = b.length + c.length;
          box.breadth = b.breadth + c.breadth;
```

```
        box.height = b.height + c.height;
        return box;
    }

    public static bool operator == (Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length == rhs.length && lhs.height == rhs.height
           && lhs.breadth == rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public static bool operator !=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length != rhs.length || lhs.height != rhs.height
            || lhs.breadth != rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public static bool operator <(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length < rhs.length && lhs.height
            < rhs.height && lhs.breadth < rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator >(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length > rhs.length && lhs.height
            > rhs.height && lhs.breadth > rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator <=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length <= rhs.length && lhs.height
            <= rhs.height && lhs.breadth <= rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator >=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length >= rhs.length && lhs.height
```

```
            >= rhs.height && lhs.breadth >= rhs.breadth)
         {
             status = true;
         }
         return status;
     }
     public override string ToString()
     {
         return String.Format("({0}, {1}, {2})", length, breadth, height);
     }

  }

  class Tester
  {
     static void Main(string[] args)
     {
       Box Box1 = new Box();           // Declare Box1 of type Box
       Box Box2 = new Box();           // Declare Box2 of type Box
       Box Box3 = new Box();           // Declare Box3 of type Box
       Box Box4 = new Box();
       double volume = 0.0;    // Store the volume of a box here

       // box 1 specification
       Box1.setLength(6.0);
       Box1.setBreadth(7.0);
       Box1.setHeight(5.0);

       // box 2 specification
       Box2.setLength(12.0);
       Box2.setBreadth(13.0);
       Box2.setHeight(10.0);

      //displaying the Boxes using the overloaded ToString():
       Console.WriteLine("Box 1: {0}", Box1.ToString());
       Console.WriteLine("Box 2: {0}", Box2.ToString());

       // volume of box 1
       volume = Box1.getVolume();
       Console.WriteLine("Volume of Box1 : {0}", volume);

       // volume of box 2
       volume = Box2.getVolume();
       Console.WriteLine("Volume of Box2 : {0}", volume);

       // Add two object as follows:
       Box3 = Box1 + Box2;
       Console.WriteLine("Box 3: {0}", Box3.ToString());
       // volume of box 3
       volume = Box3.getVolume();
       Console.WriteLine("Volume of Box3 : {0}", volume);

       //comparing the boxes
       if (Box1 > Box2)
         Console.WriteLine("Box1 is greater than Box2");
       else
         Console.WriteLine("Box1 is  greater than Box2");
       if (Box1 < Box2)
         Console.WriteLine("Box1 is less than Box2");
       else
         Console.WriteLine("Box1 is not less than Box2");
```

```
        if (Box1 >= Box2)
          Console.WriteLine("Box1 is greater or equal to Box2");
        else
          Console.WriteLine("Box1 is not greater or equal to Box2");
        if (Box1 <= Box2)
          Console.WriteLine("Box1 is less or equal to Box2");
        else
          Console.WriteLine("Box1 is not less or equal to Box2");
        if (Box1 != Box2)
          Console.WriteLine("Box1 is not equal to Box2");
        else
          Console.WriteLine("Box1 is not greater or equal to Box2");
        Box4 = Box3;
        if (Box3 == Box4)
          Console.WriteLine("Box3 is equal to Box4");
        else
          Console.WriteLine("Box3 is not equal to Box4");

        Console.ReadKey();
      }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Box 1: (6, 7, 5)
Box 2: (12, 13, 10)
Volume of Box1 : 210
Volume of Box2 : 1560
Box 3: (18, 20, 15)
Volume of Box3 : 5400
Box1 is not greater than Box2
Box1 is less than Box2
Box1 is not greater or equal to Box2
Box1 is less or equal to Box2
Box1 is not equal to Box2
Box3 is equal to Box4
```

CHAPTER

23

# C# Interfaces

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

Interfaces define properties, methods and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

## Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

```
public interface ITransactions
{
   // interface members
   void showTransaction();
   double getAmount();
}
```

### Example

The following example demonstrates implementation of the above interface:

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceApplication
{

   public interface ITransactions
   {
      // interface members
      void showTransaction();
      double getAmount();
   }
```

```
    public class Transaction : ITransactions
    {
        private string tCode;
        private string date;
        private double amount;
        public Transaction()
        {
            tCode = " ";
            date = " ";
            amount = 0.0;
        }
        public Transaction(string c, string d, double a)
        {
            tCode = c;
            date = d;
            amount = a;
        }
        public double getAmount()
        {
            return amount;
        }
        public void showTransaction()
        {
            Console.WriteLine("Transaction: {0}", tCode);
            Console.WriteLine("Date: {0}", date);
            Console.WriteLine("Amount: {0}", getAmount());
        }
    }
    class Tester
    {
        static void Main(string[] args)
        {
            Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
            Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
            t1.showTransaction();
            t2.showTransaction();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

CHAPTER

24

# C# Namespaces

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace will not conflict with the same class names declared in another.

## Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
namespace_name.item_name;
```

The following program demonstrates use of namespaces:

```
using System;
namespace first_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
```

```
{
   static void Main(string[] args)
   {
      first_space.namespace_cl fc = new first_space.namespace_cl();
      second_space.namespace_cl sc = new second_space.namespace_cl();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Inside first_space
Inside second_space
```

## The *using* Keyword

The **using** keyword states that the program is using the names in the given namespace. For example, we are using the **System** namespace in our programs. The class Console is defined there. We just write:

```
Console.WriteLine ("Hello there");
```

We could have written the fully qualified name as:

```
System.Console.WriteLine("Hello there");
```

You can also avoid prepending of namespaces with the **using** namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

Let us rewrite our preceding example, with using directive:

```
using System;
using first_space;
using second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class efg
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
   }
}
class TestClass
{
   static void Main(string[] args)
```

```
   {
      abc fc = new abc();
      efg sc = new efg();
      fc.func();
      sc.func();
      Console.ReadKey();
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Inside first_space
Inside second_space
```

# Nested Namespaces

Namespaces can be nested where you can define one namespace inside another namespace as follows:

```
namespace namespace_name1
{
   // code declarations
   namespace namespace_name2
   {
     // code declarations
   }
}
```

You can access members of nested namespace by using the dot (.) operator as follows:

```
using System;
using first_space;
using first_space.second_space;

namespace first_space
{
   class abc
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
   namespace second_space
   {
      class efg
      {
         public void func()
         {
            Console.WriteLine("Inside second_space");
         }
      }
   }
}

class TestClass
{
   static void Main(string[] args)
   {
      abc fc = new abc();
```

```
        efg sc = new efg();
        fc.func();
        sc.func();
        Console.ReadKey();
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Inside first_space
Inside second_space
```

CHAPTER

25

# C# Preprocessor Directives

The preprocessors directives give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).

C# compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In C# the preprocessor directives are used to help in conditional compilation. Unlike C and C++ directives, they are not used to create macros. A preprocessor directive must be the only instruction on a line.

## List of Preprocessor Directives in C#

The following table lists the preprocessor directives available in C#:

| Preprocessor Directive | Description. |
|---|---|
| #define | It defines a sequence of characters, called symbol. |
| #undef | It allows you to undefine a symbol. |
| #if | It allows testing a symbol or symbols to see if they evaluate to true. |
| #else | It allows to create a compound conditional directive, along with #if. |
| #elif | It allows creating a compound conditional directive. |
| #endif | Specifies the end of a conditional directive. |
| #line | It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings. |
| #error | It allows generating an error from a specific location in your code. |
| #warning | It allows generating a level one warning from a specific location in your code. |
| #region | It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. |
| #endregion | It marks the end of a #region block. |

## The #define Preprocessor

The #define preprocessor directive creates symbolic constants.

#define lets you define a symbol, such that, by using the symbol as the expression passed to the #if directive, the expression will evaluate to true. Its syntax is as follows:

```
#define symbol
```

The following program illustrates this:

```
#define PI
using System;
namespace PreprocessorDAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         #if (PI)
            Console.WriteLine("PI is defined");
         #else
            Console.WriteLine("PI is not defined");
         #endif
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
PI is defined
```

## Conditional Directives

You can use the #if directive to create a conditional directive. Conditional directives are useful for testing a symbol or symbols to see if they evaluate to true. If they do evaluate to true, the compiler evaluates all the code between the #if and the next directive.

Syntax for conditional directive is:

```
#if symbol [operator symbol]...
```

Where, *symbol* is the name of the symbol you want to test. You can also use true and false or prepend the symbol with the negation operator.

The *operator symbol* is the operator used for evaluating the symbol. Operators could be either of the following:

- == (equality)
- != (inequality)
- && (and)
- || (or)

You can also group symbols and operators with parentheses. Conditional directives are used for compiling code for a debug build or when compiling for a specific configuration. A conditional directive beginning with a **#if** directive must explicitly be terminated with a **#endif** directive.

The following program demonstrates use of conditional directives:

```
#define DEBUG
#define VC_V10
using System;
public class TestClass
{
   public static void Main()
   {

      #if (DEBUG && !VC_V10)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V10)
         Console.WriteLine("VC_V10 is defined");
      #elif (DEBUG && VC_V10)
         Console.WriteLine("DEBUG and VC_V10 are defined");
      #else
         Console.WriteLine("DEBUG and VC_V10 are not defined");
      #endif
      Console.ReadKey();
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
DEBUG and VC_V10 are defined
```

CHAPTER

26

# C# Regular Expressions

A **regular expression** is a pattern that could be matched against an input text. The .Net framework provides a regular expression engine that allows such matching. A pattern consists of one or more character literals, operators, or constructs.

## Constructs for Defining Regular Expressions

There are various categories of characters, operators, and constructs that lets you to define regular expressions. Click the follwoing links to find these constructs.

- Character escapes
- Character classes
- Anchors
- Grouping constructs
- Quantifiers
- Backreference constructs
- Alternation constructs
- Substitutions
- Miscellaneous constructs

### Character escapes

These are basically the special characters or escape characters. The backslash character (\) in a regular expression indicates that the character that follows it either is a special character or should be interpreted literally.

The following table lists the escape characters:

| Escaped character | Description | Pattern | Matches |
|---|---|---|---|
| **\a** | Matches a bell character, \u0007. | \a | "\u0007" in "Warning!" + '\u0007' |
| **\b** | In a character class, matches a backspace, \u0008. | [\b]{3,} | "\b\b\b\b" in "\b\b\b\b" |
| **\t** | Matches a tab, \u0009. | (\w+)\t | "Name\t", "Addr\t" in "Name\tAddr\t" |
| **\r** | Matches a carriage return, \u000D. (\r is not equivalent to the newline character, \n.) | \r\n(\w+) | "\r\nHello" in "\r\Hello\nWorld." |

| | | | |
|---|---|---|---|
| **\v** | Matches a vertical tab, \u000B. | [\v]{2,} | "\v\v\v" in "\v\v\v" |
| **\f** | Matches a form feed, \u000C. | [\f]{2,} | "\f\f\f" in "\f\f\f" |
| **\n** | Matches a new line, \u000A. | \r\n(\w+) | "\r\nHello" in "\r\Hello\nWorld." |
| **\e** | Matches an escape, \u001B. | \e | "\x001B" in "\x001B" |
| **\ nnn** | Uses octal representation to specify a character (nnn consists of up to three digits). | \w\040\w | "a b", "c d" in "a bc d" |
| **\x nn** | Uses hexadecimal representation to specify a character (nn consists of exactly two digits). | \w\x20\w | "a b", "c d" in "a bc d" |
| **\c X \c x** | Matches the ASCII control character that is specified by X or x, where X or x is the letter of the control character. | \cC | "\x0003" in "\x0003" (Ctrl-C) |
| **\u nnnn** | Matches a Unicode character by using hexadecimal representation (exactly four digits, as represented by nnnn). | \w\u0020\w | "a b", "c d" in "a bc d" |
| **** | When followed by a character that is not recognized as an escaped character, matches that character. | \d+[\+-x*]\d+\d+[\+-x*\d+ | "2+2" and "3*9" in "(2+2) * 3*9" |

## Character classes

A character class matches any one of a set of characters. The following table describes the character classes:

| **Character class** | **Description** | **Pattern** | **Matches** |
|---|---|---|---|
| **[character_group]** | Matches any single character in character_group. By default, the match is case-sensitive. | [mn] | "m" in "mat" "m", "n" in "moon" |
| **[^character_group]** | Negation: Matches any single character that is not in character_group. By default, characters incharacter_group are case-sensitive. | [^aei] | "v", "l" in "avail" |
| **[ first - last ]** | Character range: Matches any single character in the range from first to last. | (\w+)\t | "Name\t", "Addr\t" in "Name\tAddr\t" |
| **.** | Wildcard: Matches any single character except \n. | a.e | "ave" in "have" "ate" in "mate" |
| **\p{ name }** | Matches any single character in the Unicode general category or named block specified by *name*. | \p{Lu} | "C", "L" in "City Lights" |
| **\P{ name }** | Matches any single character that is not in the Unicode general category or named block specified by *name*. | \P{Lu} | "i", "t", "y" in "City" |
| **\w** | Matches any word character. | \w | "R", "o", "m" and "1" in "Room#1" |
| **\W** | Matches any non-word character. | \W | "#" in "Room#1" |
| **\s** | Matches any white-space character. | \w\s | "D " in "ID A1.3" |
| **\S** | Matches any non-white-space character. | \s\S | " _" in "int __ctr" |
| **\d** | Matches any decimal digit. | \d | "4" in "4 = IV" |

| | | | |
|---|---|---|---|
| **\D** | Matches any character other than a decimal digit. | \D | " ", "=", " ", "I", "V" in "4 = IV" |

## Anchors

Anchors allow a match to succeed or fail depending on the current position in the string. The following table lists the anchors:

| Assertion | Description | Pattern | Matches |
|---|---|---|---|
| **^** | The match must start at the beginning of the string or line. | ^\d{3} | "567" in "567-777-" |
| **$** | The match must occur at the end of the string or before **\n** at the end of the line or string. | -\d{4}$ | "-2012" in "8-12-2012" |
| **\A** | The match must occur at the start of the string. | \A\w{3} | "Code" in "Code-007-" |
| **\Z** | The match must occur at the end of the string or before \n at the end of the string. | -\d{3}\Z | "-007" in "Bond-901-007" |
| **\z** | The match must occur at the end of the string. | -\d{3}\z | "-333" in "-901-333" |
| **\G** | The match must occur at the point where the previous match ended. | \\G\(\d\) | "(1)", "(3)", "(5)" in "(1)(3)(5)[7](9)" |
| **\b** | The match must occur on a boundary between a **\w** (alphanumeric) and a**\W**(nonalphanumeric) character. | \w | "R", "o", "m" and "1" in "Room#1" |
| **\B** | The match must not occur on a **\b**boundary. | \Bend\w*\b | "ends", "ender" in "end sends endure lender" |

## Grouping constructs

Grouping constructs delineate sub-expressions of a regular expression and capture substrings of an input string. The following table lists the grouping constructs:

| Grouping construct | Description | Pattern | Matches |
|---|---|---|---|
| **( subexpression )** | Captures the matched subexpression and assigns it a zero-based ordinal number. | (\w)\1 | "ee" in "deep" |
| **(?< name >subexpression)** | Captures the matched subexpression into a named group. | (?< double>\w)\k< double> | "ee" in "deep" |
| **(?< name1 -name2 >subexpression)** | Defines a balancing group definition. | (((?'Open'\()[^\(\)]*)+((?'Close-Open'\))[^\(\)]*)+)*(?(Open)(?!))$ | "((1-3)*(3-1))" in "3+2^((1-3)*(3-1))" |
| **(?: subexpression)** | Defines a noncapturing group. | Write(?:Line)? | "WriteLine" in "Console.WriteLine()" |
| **(?imnsx-imnsx:subexpression)** | Applies or disables the specified options within*subexpression*. | A\d{2}(?i:\w+)\b | "A12xl", "A12XL" in "A12xl A12XL a12xl" |

| | | | |
|---|---|---|---|
| **(?= subexpression)** | Zero-width positive lookahead assertion. | \w+(?=\.) | "is", "ran", and "out" in "He is. The dog ran. The sun is out." |
| **(?! subexpression)** | Zero-width negative lookahead assertion. | \b(?!un)\w+\b | "sure", "used" in "unsure sure unity used" |
| **(?< =subexpression)** | Zero-width positive lookbehind assertion. | (?< =19)\d{2}\b | "51", "03" in "1851 1999 1950 1905 2003" |
| **(?< ! subexpression)** | Zero-width negative lookbehind assertion. | (?< !19)\d{2}\b | "ends", "ender" in "end sends endure lender" |
| **(?> subexpression)** | Nonbacktracking (or "greedy") subexpression. | [13579](?>A+B+) | "1ABB", "3ABB", and "5AB" in "1ABB 3ABBC 5AB 5AC" |

## Quantifiers

Quantifiers specify how many instances of the previous element (which can be a character, a group, or a character class) must be present in the input string for a match to occur.

| Quantifier | Description | Pattern | Matches |
|---|---|---|---|
| * | Matches the previous element zero or more times. | \d*\.\d | ".0", "19.9", "219.9" |
| + | Matches the previous element one or more times. | "be+" | "bee" in "been", "be" in "bent" |
| ? | Matches the previous element zero or one time. | "rai?n" | "ran", "rain" |
| { n } | Matches the previous element exactly n times. | ",\d{3}" | ",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210" |
| { n ,} | Matches the previous element at least n times. | "\d{2,}" | "166", "29", "1930" |
| { n , m } | Matches the previous element at least n times, but no more than m times. | "\d{3,5}" | "166", "17668" "19302" in "193024" |
| *? | Matches the previous element zero or more times, but as few times as possible. | \d*?\.\d | ".0", "19.9", "219.9" |
| +? | Matches the previous element one or more times, but as few times as possible. | "be+?" | "be" in "been", "be" in "bent" |
| ?? | Matches the previous element zero or one time, but as few times as possible. | "rai??n" | "ran", "rain" |
| { n }? | Matches the preceding element exactly n times. | ",\d{3}?" | ",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210" |
| { n ,}? | Matches the previous element at least n times, but as few times as possible. | "\d{2,}?" | "166", "29", "1930" |
| { n , m }? | Matches the previous element between n and m times, but as few times as possible. | "\d{3,5}?" | "166", "17668" "193", "024" in "193024" |

## Backreference constructs

Backreference constructs allow a previously matched sub-expression to be identified subsequently in the same regular expression.

The following table lists these constructs:

| Backreference construct | Description | Pattern | Matches |
|---|---|---|---|
| **\ number** | Backreference. Matches the value of a numbered subexpression. | (\w)\1 | "ee" in "seek" |
| **\k< name >** | Named backreference. Matches the value of a named expression. | (?< char>\w)\k< char> | "ee" in "seek" |

## Alternation constructs

Alternation constructs modify a regular expression to enable either/or matching. The following table lists the alternation constructs:

| Alternation construct | Description | Pattern | Matches |
|---|---|---|---|
| **\|** | Matches any one element separated by the vertical bar (\|) character. | th(e\|is\|at) | "the", "this" in "this is the day. " |
| **(?( expression )yes \| no )** | Matches *yes* if expression matches; otherwise, matches the optional *no*part. Expression is interpreted as a zero-width assertion. | (?(A)A\d{2}\b\|\b\d{3}\b) | "A10", "910" in "A10 C103 910" |
| **(?( name )yes \| no )** | Matches *yes* if the named capture name has a match; otherwise, matches the optional *no*. | (?< quoted>")?(?(quoted).+?"\|\S+\s) | Dogs.jpg, "Yiska playing.jpg" in "Dogs.jpg "Yiska playing.jpg"" |

## Substitution

Substitutions are used in replacement patterns. The following table lists the substitutions:

| Character | Description | Pattern | Replacement pattern | Input string | Result string |
|---|---|---|---|---|---|
| **$**number | Substitutes the substring matched by group number. | \b(\w+)(\s)(\w+)\b | $3$2$1 | "one two" | "two one" |
| **${**name**}** | Substitutes the substring matched by the named*groupname*. | \b(?< word1>\w+)(\s)(?< word2>\w+)\b | ${word2} ${word1} | "one two" | "two one" |
| **$$** | Substitutes a literal "$". | \b(\d+)\s?USD | $$$1 | "103 USD" | "$103" |
| **$&** | Substitutes a copy of the whole match. | (\$*(\d*(\.+\d+)?){1}) | **$& | "$1.30" | "**$1.30**" |

| | | | | | |
|---|---|---|---|---|---|
| **$`** | Substitutes all the text of the input string before the match. | B+ | $` | "AABBCC" | "AAAACC" |
| **$'** | Substitutes all the text of the input string after the match. | B+ | $' | "AABBCC" | "AACCCC" |
| **$+** | Substitutes the last group that was captured. | B+(C+) | $+ | "AABBCCDD" | AACCDD |
| **$_** | Substitutes the entire input string. | B+ | $_ | "AABBCC" | "AAAABBCCCC" |

## Miscellaneous constructs

Following are various miscellaneous constructs:

| Construct | Definition | Example |
|---|---|---|
| **(?imnsx-imnsx)** | Sets or disables options such as case insensitivity in the middle of a pattern. | \bA(?i)b\w+\b matches "ABA", "Able" in "ABA Able Act" |
| **(?#comment)** | Inline comment. The comment ends at the first closing parenthesis. | \bA(?#Matches words starting with A)\w+\b |
| # [to end of line] | X-mode comment. The comment starts at an unescaped # and continues to the end of the line. | (?x)\bA\w+\b#Matches words starting with A |

## The Regex Class

The Regex class is used for representing a regular expression.

The Regex class has the following commonly used methods:

| S.N | Methods & Description |
|---|---|
| 1 | **public bool IsMatch( string input )**<br>Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string. |
| 2 | **public bool IsMatch( string input, int startat )**<br>Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string. |
| 3 | **public static bool IsMatch( string input, string pattern )**<br>Indicates whether the specified regular expression finds a match in the specified input string. |
| 4 | **public MatchCollection Matches( string input )**<br>Searches the specified input string for all occurrences of a regular expression. |
| 5 | **public string Replace( string input, string replacement )**<br>In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string. |
| 6 | **public string[] Split( string input )**<br>Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor. |

For the complete list of methods and properties, please read the Microsoft documentation on C#.

## Example 1

The following example matches words that start with 'S':

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "A Thousand Splendid Suns";

         Console.WriteLine("Matching words that start with 'S': ");
         showMatch(str, @"\bS\S*");
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns
```

## Example 2

The following example matches words that start with 'm' and ends with 'e':

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
```

```
        {
            string str = "make maze and manage to measure it";

            Console.WriteLine("Matching words start with 'm' and ends with 'e':");
            showMatch(str, @"\bm\S*e\b");
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```

## Example 3

This example replaces extra white space:

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string input = "Hello    World    ";
            string pattern = "\\s+";
            string replacement = " ";
            Regex rgx = new Regex(pattern);
            string result = rgx.Replace(input, replacement);

            Console.WriteLine("Original String: {0}", input);
            Console.WriteLine("Replacement String: {0}", result);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Original String: Hello    World
Replacement String: Hello World
```

CHAPTER

27

# C# Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally** and **throw**.

- **try**: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- **catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally**: The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw**: A program throws an exception when a problem shows up. This is done using a throw keyword.

## Syntax

Assuming a block will raise and exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   // statements causing exception
}
catch( ExceptionName e1 )
{
   // error handling code
}
catch( ExceptionName e2 )
{
   // error handling code
}
catch( ExceptionName eN )
{
   // error handling code
}
finally
{
   // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

# Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.
The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class.
The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class:

| Exception Class | Description |
|---|---|
| System.IO.IOException | Handles I/O errors. |
| System.IndexOutOfRangeException | Handles errors generated when a method refers to an array index out of range. |
| System.ArrayTypeMismatchException | Handles errors generated when type is mismatched with the array type. |
| System.NullReferenceException | Handles errors generated from deferencing a null object. |
| System.DivideByZeroException | Handles errors generated from dividing a dividend with zero. |
| System.InvalidCastException | Handles errors generated during typecasting. |
| System.OutOfMemoryException | Handles errors generated from insufficient free memory. |
| System.StackOverflowException | Handles errors generated from stack overflow. |

# Handling Exceptions

C# provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch** and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs:

```
using System;
namespace ErrorHandlingApplication
{
    class DivNumbers
    {
        int result;
        DivNumbers()
        {
            result = 0;
        }
        public void division(int num1, int num2)
        {
            try
            {
                result = num1 / num2;
            }
            catch (DivideByZeroException e)
            {
```

```
                Console.WriteLine("Exception caught: {0}", e);
            }
            finally
            {
                Console.WriteLine("Result: {0}", result);
            }

        }
        static void Main(string[] args)
        {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

# Creating User-Defined Exceptions

You can also define your own exception. User-defined exception classes are derived from the**ApplicationException** class. The following example demonstrates this:

```
using System;
namespace UserDefinedException
{
   class TestTemperature
   {
      static void Main(string[] args)
      {
         Temperature temp = new Temperature();
         try
         {
            temp.showTemp();
         }
         catch(TempIsZeroException e)
         {
            Console.WriteLine("TempIsZeroException: {0}", e.Message);
         }
         Console.ReadKey();
      }
   }
}
public class TempIsZeroException: ApplicationException
{
   public TempIsZeroException(string message): base(message)
   {
   }
}
public class Temperature
{
   int temperature = 0;
   public void showTemp()
   {
      if(temperature == 0)
```

```
        {
           throw (new TempIsZeroException("Zero Temperature found"));
        }
        else
        {
           Console.WriteLine("Temperature: {0}", temperature);
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
TempIsZeroException: Zero Temperature found
```

# Throwing Objects

You can throw an object if it is either directly or indirectly derived from the **System.Exception** class. You can use a throw statement in the catch block to throw the present object as:

```
Catch(Exception e)
{
   ...
   Throw e
}
```

CHAPTER

28

# C# File I/O

A file is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a **stream**.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the **input stream** and the **output stream**. The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

## C# I/O Classes

The System.IO namespace has various class that are used for performing various operation with files, like creating and deleting files, reading from or writing to a file, closing a file etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace:

| I/O Class | Description |
|---|---|
| BinaryReader | Reads primitive data from a binary stream. |
| BinaryWriter | Writes primitive data in binary format. |
| BufferedStream | A temporary storage for a stream of bytes. |
| Directory | Helps in manipulating a directory structure. |
| DirectoryInfo | Used for performing operations on directories. |
| DriveInfo | Provides information for the drives. |
| File | Helps in manipulating files. |
| FileInfo | Used for performing operations on files. |
| FileStream | Used to read from and write to any location in a file. |
| MemoryStream | Used for random access to streamed data stored in memory. |
| Path | Performs operations on path information. |
| StreamReader | Used for reading characters from a byte stream. |

| | |
|---|---|
| StreamWriter | Is used for writing characters to a stream. |
| StringReader | Is used for reading from a string buffer. |
| StringWriter | Is used for writing into a string buffer. |

# The FileStream Class

The **FileStream** class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

You need to create a **FileStream** object to create a new file or open an existing file. The syntax for creating a **FileStream** object is as follows:

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
```

For example, for creating a FileStream object **F** for reading a file named **sample.txt**:

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read,
FileShare.Read);
```

| Parameter | Description |
|---|---|
| FileMode | The **FileMode** enumerator defines various methods for opening files. The members of the FileMode enumerator are:<br>**Append**: It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist.<br>**Create**: It creates a new file.<br>**CreateNew**: It specifies to the operating system, that it should create a new file.<br>**Open**: It opens an existing file.<br>**OpenOrCreate**: It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file.<br>**Truncate**: It opens an existing file and truncates its size to zero bytes. |
| FileAccess | **FileAccess** enumerators have members: **Read**, **ReadWrite** and **Write**. |
| FileShare | **FileShare** enumerators have the following members:<br>**Inheritable**: It allows a file handle to pass inheritance to the child processes<br>**None**: It declines sharing of the current file<br>**Read**: It allows opening the file for reading<br>**ReadWrite**: It allows opening the file for reading and writing<br>**Write**: It allows opening the file for writing |

## Example:

The following program demonstrates use of the **FileStream** class:

```
using System;
using System.IO;

namespace FileIOApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream F = new FileStream("test.dat",
```

```
                FileMode.OpenOrCreate, FileAccess.ReadWrite);

                for (int i = 1; i <= 20; i++)
                {
                    F.WriteByte((byte)i);
                }

                F.Position = 0;

                for (int i = 0; i <= 20; i++)
                {
                    Console.Write(F.ReadByte() + " ");
                }
                F.Close();
                Console.ReadKey();
            }
        }
}
```

When the above code is compiled and executed, it produces the following result:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

# Advanced File Operations in C#

The preceding example provides simple file operations in C#. However, to utilize the immense powers of C# System.IO classes, you need to know the commonly used properties and methods of these classes.

We will discuss these classes and the operations they perform, in the following sections. Please click the links provided to get to the individual sections:

| Topic and Description |
|---|
| Reading from and Writing into Text files<br>It involves reading from and writing into text files. The **StreamReader** and **StreamWriter** class helps to accomplish it. |
| Reading from and Writing into Binary files<br>It involves reading from and writing into binary files. The **BinaryReader** and **BinaryWriter** class helps to accomplish this. |
| Manipulating the Windows file system<br>It gives a C# programamer the ability to browse and locate Windows files and directories. |

# Reading from and Writing into Text files

The **StreamReader** and **StreamWriter** classes are used for reading from and writing data to text files. These classes inherit from the abstract base class Stream, which supports reading and writing bytes into a file stream.

## The StreamReader Class

The **StreamReader** class also inherits from the abstract base class TextReader that represents a reader for reading series of characters. The following table describes some of the commonly used**methods** of the StreamReader class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public override void Close()**<br>It closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader. |
| 2 | **public override int Peek()**<br>Returns the next available character but does not consume it. |
| 3 | **public override int Read()**<br>Reads the next character from the input stream and advances the character position by one character. |

## Example:

The following example demonstrates reading a text file named Jamaica.txt. The file reads:

```
Down the way where the nights are gay
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

```
using System;
using System.IO;

namespace FileApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Create an instance of StreamReader to read from a file.
                // The using statement also closes the StreamReader.
                using (StreamReader sr = new StreamReader("c:/jamaica.txt"))
                {
                    string line;

                    // Read and display lines from the file until
                    // the end of the file is reached.
                    while ((line = sr.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }
            }
            catch (Exception e)
            {
                // Let the user know what went wrong.
                Console.WriteLine("The file could not be read:");
                Console.WriteLine(e.Message);
            }
            Console.ReadKey();
        }
    }
}
```

Guess what it displays when you compile and run the program!

## The StreamWriter Class

The **StreamWriter** class inherits from the abstract class TextWriter that represents a writer, which can write a series of character.

The following table shows some of the most commonly used methods of this class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public override void Close()**<br>Closes the current StreamWriter object and the underlying stream. |
| 2 | **public override void Flush()**<br>Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream. |
| 3 | **public virtual void Write(bool value)**<br>Writes the text representation of a Boolean value to the text string or stream. (Inherited from TextWriter.) |
| 4 | **public override void Write( char value )**<br>Writes a character to the stream. |
| 5 | **public virtual void Write( decimal value )**<br>Writes the text representation of a decimal value to the text string or stream. |
| 6 | **public virtual void Write( double value )**<br>Writes the text representation of an 8-byte floating-point value to the text string or stream. |
| 7 | **public virtual void Write( int value )**<br>Writes the text representation of a 4-byte signed integer to the text string or stream. |
| 8 | **public override void Write( string value )**<br>Writes a string to the stream. |
| 9 | **public virtual void WriteLine()**<br>Writes a line terminator to the text string or stream. |

For complete list of methods, please visit Microsoft's C# documentation.

## Example:

The following example demonstrates writing text data into a file using the StreamWriter class:

```
using System;
using System.IO;

namespace FileApplication
{
    class Program
    {
        static void Main(string[] args)
        {

            string[] names = new string[] {"Zara Ali", "Nuha Ali"};
            using (StreamWriter sw = new StreamWriter("names.txt"))
            {
                foreach (string s in names)
                {
                    sw.WriteLine(s);

                }
            }
```

```
            // Read and show each line from the file.
            string line = "";
            using (StreamReader sr = new StreamReader("names.txt"))
            {
                while ((line = sr.ReadLine()) != null)
                {
                    Console.WriteLine(line);
                }
            }
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Zara Ali
Nuha Ali
```

# Reading from and Writing into Binary files

The **BinaryReader** and **BinaryWriter** classes are used for reading from and writing to a binary file.

## The BinaryReader Class

The **BinaryReader** class is used to read binary data from a file. A **BinaryReader** object is created by passing a **FileStream** object to its constructor.

The following table shows some of the commonly used **methods** of the **BinaryReader** class.

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public override void Close()**<br>It closes the BinaryReader object and the underlying stream. |
| 2 | **public virtual int Read()**<br>Reads the characters from the underlying stream and advances the current position of the stream. |
| 3 | **public virtual bool ReadBoolean()**<br>Reads a Boolean value from the current stream and advances the current position of the stream by one byte. |
| 4 | **public virtual byte ReadByte()**<br>Reads the next byte from the current stream and advances the current position of the stream by one byte. |
| 5 | **public virtual byte[] ReadBytes( int count )**<br>Reads the specified number of bytes from the current stream into a byte array and advances the current position by that number of bytes. |
| 6 | **public virtual char ReadChar()**<br>Reads the next character from the current stream and advances the current position of the stream in accordance with the Encoding used and the specific character being read from the stream. |
| 7 | **public virtual char[] ReadChars( int count )**<br>Reads the specified number of characters from the current stream, returns the data in a character array, and advances the current position in accordance with the Encoding used and the specific character being read from the stream. |
| 8 | **public virtual double ReadDouble()**<br>Reads an 8-byte floating point value from the current stream and advances the current position of the stream by eight bytes. |

| | |
|---|---|
| 9 | **public virtual int ReadInt32()**<br>Reads a 4-byte signed integer from the current stream and advances the current position of the stream by four bytes. |
| 10 | **public virtual string ReadString()**<br>Reads a string from the current stream. The string is prefixed with the length, encoded as an integer seven bits at a time. |

## The BinaryWriter Class

The **BinaryWriter** class is used to write binary data to a stream. A BinaryWriter object is created by passing a FileStream object to its constructor.

The following table shows some of the commonly used methods of the BinaryWriter class.

| S.N | Function Name & Description |
|---|---|
| 1 | **public override void Close()**<br>It closes the BinaryWriter object and the underlying stream. |
| 2 | **public virtual void Flush()**<br>Clears all buffers for the current writer and causes any buffered data to be written to the underlying device. |
| 3 | **public virtual long Seek( int offset, SeekOrigin origin )**<br>Sets the position within the current stream. |
| 4 | **public virtual void Write( bool value )**<br>Writes a one-byte Boolean value to the current stream, with 0 representing false and 1 representing true. |
| 5 | **public virtual void Write( byte value )**<br>Writes an unsigned byte to the current stream and advances the stream position by one byte. |
| 6 | **public virtual void Write( byte[] buffer )**<br>Writes a byte array to the underlying stream. |
| 7 | **public virtual void Write( char ch )**<br>Writes a Unicode character to the current stream and advances the current position of the stream in accordance with the Encoding used and the specific characters being written to the stream. |
| 8 | **public virtual void Write( char[] chars )**<br>Writes a character array to the current stream and advances the current position of the stream in accordance with the Encoding used and the specific characters being written to the stream. |
| 9 | **public virtual void Write( double value )**<br>Writes an eight-byte floating-point value to the current stream and advances the stream position by eight bytes. |
| 10 | **public virtual void Write( int value )**<br>Writes a four-byte signed integer to the current stream and advances the stream position by four bytes. |
| 11 | **public virtual void Write( string value )**<br>Writes a length-prefixed string to this stream in the current encoding of the BinaryWriter, and advances the current position of the stream in accordance with the encoding used and the specific characters being written to the stream. |

For complete list of methods, please visit Microsoft's C# documentation.

## Example

The following example demonstrates reading and writing binary data:

```
using System;
```

```
using System.IO;

namespace BinaryFileApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            BinaryWriter bw;
            BinaryReader br;
            int i = 25;
            double d = 3.14157;
            bool b = true;
            string s = "I am happy";
            //create the file
            try
            {
                bw = new BinaryWriter(new FileStream("mydata",
                                  FileMode.Create));
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot create file.");
                return;
            }
            //writing into the file
            try
            {
                bw.Write(i);
                bw.Write(d);
                bw.Write(b);
                bw.Write(s);
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot write to file.");
                return;
            }

            bw.Close();
            //reading from the file
            try
            {
                br = new BinaryReader(new FileStream("mydata",
                                  FileMode.Open));
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot open file.");
                return;
            }
            try
            {
                i = br.ReadInt32();
                Console.WriteLine("Integer data: {0}", i);
                d = br.ReadDouble();
                Console.WriteLine("Double data: {0}", d);
                b = br.ReadBoolean();
                Console.WriteLine("Boolean data: {0}", b);
                s = br.ReadString();
                Console.WriteLine("String data: {0}", s);
```

```
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot read from file.");
                return;
            }
            br.Close();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Integer data: 25
Double data: 3.14157
Boolean data: True
String data: I am happy
```

# Manipulating the Windows file system

C# allows you to work with the directories and files using various directory and file related classes like, the **DirectoryInfo** class and the **FileInfo** class.

## The DirectoryInfo Class

The **DirectoryInfo** class is derived from the **FileSystemInfo** class. It has various methods for creating, moving, and browsing through directories and subdirectories. This class cannot be inherited.

Following are some commonly used **properties** of the **DirectoryInfo** class:

| S.N | Property Name & Description |
|---|---|
| 1 | **Attributes**<br>Gets the attributes for the current file or directory. |
| 2 | **CreationTime**<br>Gets the creation time of the current file or directory. |
| 3 | **Exists**<br>Gets a Boolean value indicating whether the directory exists. |
| 4 | **Extension**<br>Gets the string representing the file extension. |
| 5 | **FullName**<br>Gets the full path of the directory or file. |
| 6 | **LastAccessTime**<br>Gets the time the current file or directory was last accessed. |
| 7 | **Name**<br>Gets the name of this DirectoryInfo instance. |

Following are some commonly used **methods** of the **DirectoryInfo** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public void Create()**<br>Creates a directory. |

| | |
|---|---|
| 2 | **public DirectoryInfo CreateSubdirectory( string path )**<br>Creates a subdirectory or subdirectories on the specified path. The specified path can be relative to this instance of the DirectoryInfo class. |
| 3 | **public override void Delete()**<br>Deletes this DirectoryInfo if it is empty. |
| 4 | **public DirectoryInfo[] GetDirectories()**<br>Returns the subdirectories of the current directory. |
| 5 | **public FileInfo[] GetFiles()**<br>Returns a file list from the current directory. |

For complete list of properties and methods please visit Microsoft's C# documentation.

## The FileInfo Class

The **FileInfo** class is derived from the **FileSystemInfo** class. It has properties and instance methods for creating, copying, deleting, moving, and opening of files, and helps in the creation of FileStream objects. This class cannot be inherited.

Following are some commonly used **properties** of the **FileInfo** class:

| S.N | Property Name & Description |
|---|---|
| 1 | **Attributes**<br>Gets the attributes for the current file. |
| 2 | **CreationTime**<br>Gets the creation time of the current file. |
| 3 | **Directory**<br>Gets an instance of the directory which the file belongs to. |
| 4 | **Exists**<br>Gets a Boolean value indicating whether the file exists. |
| 5 | **Extension**<br>Gets the string representing the file extension. |
| 6 | **FullName**<br>Gets the full path of the file. |
| 7 | **LastAccessTime**<br>Gets the time the current file was last accessed. |
| 8 | **LastWriteTime**<br>Gets the time of the last written activity of the file. |
| 9 | **Length**<br>Gets the size, in bytes, of the current file. |
| 10 | **Name**<br>Gets the name of the file. |

Following are some commonly used **methods** of the **FileInfo** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public StreamWriter AppendText()**<br>Creates a StreamWriter that appends text to the file represented by this instance of the FileInfo. |

| | |
|---|---|
| 2 | **public FileStream Create()**<br>Creates a file. |
| 3 | **public override void Delete()**<br>Deletes a file permanently. |
| 4 | **public void MoveTo( string destFileName )**<br>Moves a specified file to a new location, providing the option to specify a new file name. |
| 5 | **public FileStream Open( FileMode mode )**<br>Opens a file in the specified mode. |
| 6 | **public FileStream Open( FileMode mode, FileAccess access )**<br>Opens a file in the specified mode with read, write, or read/write access. |
| 7 | **public FileStream Open( FileMode mode, FileAccess access, FileShare share )**<br>Opens a file in the specified mode with read, write, or read/write access and the specified sharing option. |
| 8 | **public FileStream OpenRead()**<br>Creates a read-only FileStream |
| 9 | **public FileStream OpenWrite()**<br>Creates a write-only FileStream. |

For complete list of properties and methods, please visit Microsoft's C# documentation

## Example

The following example demonstrates the use of the above-mentioned classes:

```
using System;
using System.IO;

namespace WindowsFileApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            //creating a DirectoryInfo object
            DirectoryInfo mydir = new DirectoryInfo(@"c:\Windows");

            // getting the files in the directory, their names and size
            FileInfo [] f = mydir.GetFiles();
            foreach (FileInfo file in f)
            {
                Console.WriteLine("File Name: {0} Size: {1}",
                                  file.Name, file.Length);
            }
            Console.ReadKey();
        }
    }
}
```

When you compile and run the program, it displays the names of files and their size in the Windows directory.

CHAPTER

29

# C# Attributes

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc., in your program. You can add declarative information to a program by using an attribute. A declarative tag is depicted by square ([ ]) brackets placed above the element it is used for.

Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. The .Net Framework provides two types of attributes: *the pre-defined* attributes and *custom built* attributes.

## Specifying an Attribute

Syntax for specifying an attribute is as follows:

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

Name of the attribute and its values are specified within the square brackets, before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

## Predefined Attributes

The .Net Framework provides three pre-defined attributes:

- AttributeUsage
- Conditional
- Obsolete

### AttributeUsage:

The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

Syntax for specifying this attribute is as follows:

```
[AttributeUsage(
   validon,
```

```
    AllowMultiple=allowmultiple,
    Inherited=inherited
)]
```

Where,

- The parameter validon specifies the language elements on which the attribute can be placed. It is a combination of the value of an enumerator *AttributeTargets*. The default value is *AttributeTargets.All*.
- The parameter *allowmultiple* (optional) provides value for the *AllowMultiple* property of this attribute, a Boolean value. If this is true, the attribute is multiuse. The default is false (single-use).
- The parameter inherited (optional) provides value for the *Inherited* property of this attribute, a Boolean value. If it is true, the attribute is inherited by derived classes. The default value is false (not inherited).

For example,

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Feild |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

## Conditional

This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

It causes conditional compilation of method calls, depending on the specified value such as **Debug** or **Trace**. For example, it displays the values of the variables while debugging a code.

Syntax for specifying this attribute is as follows:

```
[Conditional(
    conditionalSymbol
)]
```

For example,

```
[Conditional("DEBUG")]
```

The following example demonstrates the attribute:

```
#define DEBUG
using System;
using System.Diagnostics;
public class Myclass
{
    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }
}
class Test
{
    static void function1()
    {
        Myclass.Message("In Function 1.");
        function2();
    }
```

```
    static void function2()
    {
        Myclass.Message("In Function 2.");
    }
    public static void Main()
    {
        Myclass.Message("In Main function.");
        function1();
        Console.ReadKey();
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
In Main function
In Function 1
In Function 2
```

## Obsolete

This predefined attribute marks a program entity that should not be used. It enables you to inform the compiler to discard a particular target element. For example, when a new method is being used in a class, but you still want to retain the old method in the class, you may mark it as obsolete by displaying a message the new method should be used, instead of the old method.

Syntax for specifying this attribute is as follows:

```
[Obsolete(
   message
)]
[Obsolete(
   message,
   iserror
)]
```

Where,

- The parameter *message*, is a string describing the reason why the item is obsolete and what to use instead.
- The parameter *iserror*, is a Boolean value. If its value is true, the compiler should treat the use of the item as an error. Default value is false (compiler generates a warning).

The following program demonstrates this:

```
using System;
public class MyClass
{
   [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
   static void OldMethod()
   {
      Console.WriteLine("It is the old method");
   }
   static void NewMethod()
   {
      Console.WriteLine("It is the new method");
   }
   public static void Main()
   {
      OldMethod();
   }
```

```
}
```

When you try to compile the program, the compiler gives an error message stating:

```
Don't use OldMethod, use NewMethod instead
```

# Creating Custom Attributes

The .Net Framework allows creation of custom attributes that can be used to store declarative information and can be retrieved at run-time. This information can be related to any target element depending upon the design criteria and application need.

Creating and using custom attributes involve four steps:

- Declaring a custom attribute
- Constructing the custom attribute
- Apply the custom attribute on a target program element
- Accessing Attributes Through Reflection

The Last step involves writing a simple program to read through the metadata to find various notations. Metadata is data about data or information used for describing other data. This program should use reflections for accessing attributes at runtime. This we will discuss in the next chapter.

## Declaring a Custom Attribute

A new custom attribute should is derived from the **System.Attribute** class. For example,

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

In the preceding code, we have declared a custom attribute named *DeBugInfo*.

## Constructing the Custom Attribute

Let us construct a custom attribute named *DeBugInfo*, which will store the information obtained by debugging any program. Let it store the following information:

- The code number for the bug
- Name of the developer who identified the bug
- Date of last review of the code
- A string message for storing the developer's remarks

Our *DeBugInfo* class will have three private properties for storing the first three information and a public property for storing the message. So the bug number, developer�s name and date of review will be the positional parameters of the DeBugInfo class and the message will be an optional or named parameter.

Each attribute must have at least one constructor. The positional parameters should be passed through the constructor. The following code shows the *DeBugInfo* class:

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
  private int bugNo;
  private string developer;
  private string lastReview;
  public string message;

  public DeBugInfo(int bg, string dev, string d)
  {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
  }

  public int BugNo
  {
      get
      {
          return bugNo;
      }
  }
  public string Developer
  {
      get
      {
          return developer;
      }
  }
  public string LastReview
  {
      get
      {
          return lastReview;
      }
  }
  public string Message
  {
      get
      {
          return message;
      }
      set
      {
          message = value;
      }
  }
```

```
}
```

## Applying the Custom Attribute

The attribute is applied by placing it immediately before its target:

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle
{
  //member variables
  protected double length;
  protected double width;
  public Rectangle(double l, double w)
  {
      length = l;
      width = w;
  }
  [DeBugInfo(55, "Zara Ali", "19/10/2012",
  Message = "Return type mismatch")]
  public double GetArea()
  {
      return length * width;
  }
  [DeBugInfo(56, "Zara Ali", "19/10/2012")]
  public void Display()
  {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
  }
}
```

In the next chapter, we will retrieve these attribute information using a Reflection class object.

CHAPTER

30

# C# Reflection

Reflection objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the **System.Reflection** namespace.

The **System.Reflection** namespace contains classes that allow you to obtain information about the application and to dynamically add types, values and objects to the application.

## Uses of Reflection

Reflection has the following uses:

- It allows view attribute information at runtime.
- It allows examining various types in an assembly and instantiate these types.
- It allows late binding to methods and properties
- It allows creating new types at runtime and then performs some tasks using those types.

## Viewing Metadata

We have mentioned in the preceding chapter that using reflection you can view the attribute information.

The **MemberInfo** object of the **System.Reflection** class need to be initialized for discovering the attributes asscociated with a class. To do this, you define an object of the target class, as:

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

The following program demonstrates this:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
   public readonly string Url;

   public string Topic  // Topic is a named parameter
   {
```

```
      get
      {
         return topic;
      }
      set
      {

         topic = value;
      }
   }

   public HelpAttribute(string url)  // url is a positional parameter
   {
      this.Url = url;
   }

   private string topic;
}
[HelpAttribute("Information on the class MyClass")]
class MyClass
{
}

namespace AttributeAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         System.Reflection.MemberInfo info = typeof(MyClass);
         object[] attributes = info.GetCustomAttributes(true);
         for (int i = 0; i < attributes.Length; i++)
         {
            System.Console.WriteLine(attributes[i]);
         }
         Console.ReadKey();

      }
   }
}
```

When it is compiled and run, it displays the name of the custom attributes attached to the class*MyClass*:

```
HelpAttribute
```

## Example

In this example, we will use the *DeBugInfo* attribute created in the previous chapter and use reflection to read metadata in the *Rectangle* class.

```
using System;
using System.Reflection;
namespace BugFixApplication
{
   //a custom attribute BugFix to be
         //assigned to a class and its members
   [AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
```

```
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
   private int bugNo;
   private string developer;
   private string lastReview;
   public string message;

   public DeBugInfo(int bg, string dev, string d)
   {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
   }

   public int BugNo
   {
      get
      {
         return bugNo;
      }
   }
   public string Developer
   {
      get
      {
         return developer;
      }
   }
   public string LastReview
   {
      get
      {
         return lastReview;
      }
   }
   public string Message
   {
      get
      {
         return message;
      }
      set
      {
         message = value;
      }
   }
}
[DeBugInfo(45, "Zara Ali", "12/8/2012",
     Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012",
     Message = "Unused variable")]
class Rectangle
{
   //member variables
   protected double length;
   protected double width;
   public Rectangle(double l, double w)
   {
```

```
         length = l;
         width = w;
      }
      [DeBugInfo(55, "Zara Ali", "19/10/2012",
           Message = "Return type mismatch")]
      public double GetArea()
      {
         return length * width;
      }
      [DeBugInfo(56, "Zara Ali", "19/10/2012")]
      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle(4.5, 7.5);
         r.Display();
         Type type = typeof(Rectangle);
         //iterating through the attribtues of the Rectangle class
         foreach (Object attributes in type.GetCustomAttributes(false))
         {
            DeBugInfo dbi = (DeBugInfo)attributes;
            if (null != dbi)
            {
               Console.WriteLine("Bug no: {0}", dbi.BugNo);
               Console.WriteLine("Developer: {0}", dbi.Developer);
               Console.WriteLine("Last Reviewed: {0}",
                                            dbi.LastReview);
               Console.WriteLine("Remarks: {0}", dbi.Message);
            }
         }

         //iterating through the method attribtues
         foreach (MethodInfo m in type.GetMethods())
         {
            foreach (Attribute a in m.GetCustomAttributes(true))
            {
               DeBugInfo dbi = (DeBugInfo)a;
               if (null != dbi)
               {
                  Console.WriteLine("Bug no: {0}, for Method: {1}",
                                                dbi.BugNo, m.Name);
                  Console.WriteLine("Developer: {0}", dbi.Developer);
                  Console.WriteLine("Last Reviewed: {0}",
                                                dbi.LastReview);
                  Console.WriteLine("Remarks: {0}", dbi.Message);
               }
            }
         }
         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Length: 4.5
Width: 7.5
Area: 33.75
Bug No: 49
Developer: Nuha Ali
Last Reviewed: 10/10/2012
Remarks: Unused variable
Bug No: 45
Developer: Zara Ali
Last Reviewed: 12/8/2012
Remarks: Return type mismatch
Bug No: 55, for Method: GetArea
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks: Return type mismatch
Bug No: 56, for Method: Display
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks:
```

CHAPTER

31

# C# Properties

Properties are named members of classes, structures and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.
Properties do not name the storage locations. Instead, they have **accessors** that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

## Accessors

The **accessor** of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both. For example:

```
// Declare a Code property of type string:
public string Code
{
   get
   {
      return code;
   }
   set
   {
      code = value;
   }
}

// Declare a Name property of type string:
public string Name
{
   get
   {
     return name;
   }
   set
   {
     name = value;
   }
}

// Declare a Age property of type int:
public int Age
```

```
{
   get
   {
      return age;
   }
   set
   {
      age = value;
   }
}
```

## Example:

The following example demonstrates use of properties:

```
using System;
class Student
{

   private string code = "N.A";
   private string name = "not known";
   private int age = 0;

   // Declare a Code property of type string:
   public string Code
   {
      get
      {
         return code;
      }
      set
      {
         code = value;
      }
   }

   // Declare a Name property of type string:
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }

   // Declare a Age property of type int:
   public int Age
   {
      get
      {
         return age;
      }
      set
      {
         age = value;
      }
   }
```

```
    public override string ToString()
    {
       return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
    }

    public static void Main()
    {
       // Create a new Student object:
       Student s = new Student();

       // Setting code, name and the age of the student
       s.Code = "001";
       s.Name = "Zara";
       s.Age = 9;
       Console.WriteLine("Student Info: {0}", s);
       //let us increase age
       s.Age += 1;
       Console.WriteLine("Student Info: {0}", s);
       Console.ReadKey();
     }
}
```

When the above code is compiled and executed, it produces the following result:

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

# Abstract Properties

An abstract class may have an abstract property, which should be implemented in the derived class. The following program illustrates this:

```
using System;
public abstract class Person
{
    public abstract string Name
    {
       get;
       set;
    }
    public abstract int Age
    {
       get;
       set;
    }
}
class Student : Person
{

    private string code = "N.A";
    private string name = "N.A";
    private int age = 0;

    // Declare a Code property of type string:
    public string Code
    {
       get
       {
          return code;
       }
```

```
      set
      {
         code = value;
      }
   }

   // Declare a Name property of type string:
   public override string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }

   // Declare a Age property of type int:
   public override int Age
   {
      get
      {
         return age;
      }
      set
      {
         age = value;
      }
   }
   public override string ToString()
   {
      return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
   }

   public static void Main()
   {
      // Create a new Student object:
      Student s = new Student();

      // Setting code, name and the age of the student
      s.Code = "001";
      s.Name = "Zara";
      s.Age = 9;
      Console.WriteLine("Student Info:- {0}", s);
      //let us increase age
      s.Age += 1;
      Console.WriteLine("Student Info:- {0}", s);
      Console.ReadKey();
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

CHAPTER

32

# C# Indexers

An **indexer** allows an object to be indexed like an array. When you define an indexer for a class, this class behaves like a **virtual array**. You can then access the instance of this class using the array access operator ([ ]).

## Syntax

A one dimensional indexer has the following syntax:

```
element-type this[int index]
{
   // The get accessor.
   get
   {
      // return the value specified by index
   }

   // The set accessor.
   set
   {
      // set the value specified by index
   }
}
```

## Use of Indexers

Declaration of behavior of an indexer is to some extent similar to a property. Like properties, you use **get** and **set** accessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.

Defining a property involves providing a property name. Indexers are not defined with names, but with the **this** keyword, which refers to the object instance. The following example demonstrates the concept:

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
```

```
        {
            for (int i = 0; i < size; i++)
            namelist[i] = "N. A.";
        }
        public string this[int index]
        {
            get
            {
                string tmp;

                if( index >= 0 && index <= size-1 )
                {
                    tmp = namelist[index];
                }
                else
                {
                    tmp = "";
                }

                return ( tmp );
            }
            set
            {
                if( index >= 0 && index <= size-1 )
                {
                    namelist[index] = value;
                }
            }
        }

        static void Main(string[] args)
        {
            IndexedNames names = new IndexedNames();
            names[0] = "Zara";
            names[1] = "Riz";
            names[2] = "Nuha";
            names[3] = "Asif";
            names[4] = "Davinder";
            names[5] = "Sunil";
            names[6] = "Rubic";
            for ( int i = 0; i < IndexedNames.size; i++ )
            {
                Console.WriteLine(names[i]);
            }
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
```

```
N. A.
```

# Overloaded Indexers

Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

The following example demonstrates overloaded indexers:

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
      {
         for (int i = 0; i < size; i++)
         {
          namelist[i] = "N. A.";
         }
      }
      public string this[int index]
      {
         get
         {
            string tmp;

            if( index >= 0 && index <= size-1 )
            {
               tmp = namelist[index];
            }
            else
            {
               tmp = "";
            }

            return ( tmp );
         }
         set
         {
            if( index >= 0 && index <= size-1 )
            {
               namelist[index] = value;
            }
         }
      }
      public int this[string name]
      {
         get
         {
            int index = 0;
            while(index < size)
            {
               if (namelist[index] == name)
               {
                return index;
```

```
            }
            index++;
         }
         return index;
      }

   }

   static void Main(string[] args)
   {
      IndexedNames names = new IndexedNames();
      names[0] = "Zara";
      names[1] = "Riz";
      names[2] = "Nuha";
      names[3] = "Asif";
      names[4] = "Davinder";
      names[5] = "Sunil";
      names[6] = "Rubic";
      //using the first indexer with int parameter
      for (int i = 0; i < IndexedNames.size; i++)
      {
         Console.WriteLine(names[i]);
      }
      //using the second indexer with the string parameter
      Console.WriteLine(names["Nuha"]);
      Console.ReadKey();
   }
  }
}
```

When the above code is compiled and executed, it produces the following result:

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

CHAPTER

33

# C# Delegates

C# delegates are similar to pointers to functions in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

## Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which have the same signature as that of the delegate.

For example, consider a delegate:

```
public delegate int MyDelegate (string s);
```

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is:

```
delegate <return type> <delegate-name> <parameter list>
```

## Instantiating Delegates

Once a delegate type has been declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written like a method call, but without the arguments to the method. For example:

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

Following example demonstrates declaration, instantiation and use of a delegate that can be used to reference methods that take an integer parameter and returns an integer value.

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
```

```
    class TestDelegate
    {
       static int num = 10;
       public static int AddNum(int p)
       {
          num += p;
          return num;
       }

       public static int MultNum(int q)
       {
          num *= q;
          return num;
       }
       public static int getNum()
       {
          return num;
       }

       static void Main(string[] args)
       {
          //create delegate instances
          NumberChanger nc1 = new NumberChanger(AddNum);
          NumberChanger nc2 = new NumberChanger(MultNum);
          //calling the methods using the delegate objects
          nc1(25);
          Console.WriteLine("Value of Num: {0}", getNum());
          nc2(5);
          Console.WriteLine("Value of Num: {0}", getNum());
          Console.ReadKey();
       }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of Num: 35
Value of Num: 175
```

# Multicasting of a Delegate

Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.

Using this useful property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The following program demonstrates multicasting of a delegate:

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
    class TestDelegate
    {
       static int num = 10;
       public static int AddNum(int p)
       {
          num += p;
```

```
         return num;
      }

      public static int MultNum(int q)
      {
         num *= q;
         return num;
      }
      public static int getNum()
      {
         return num;
      }

      static void Main(string[] args)
      {
         //create delegate instances
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultNum);
         nc = nc1;
         nc += nc2;
         //calling multicast
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of Num: 75
```

# Use of Delegate

The following example demonstrates the use of delegate. The delegate *printString* can be used to reference methods that take a string as input and return nothing.

We use this delegate to call two methods, the first prints the string to the console, and the second one prints it to a file:

```
using System;
using System.IO;

namespace DelegateAppl
{
   class PrintString
   {
      static FileStream fs;
      static StreamWriter sw;
      // delegate declaration
      public delegate void printString(string s);

      // this method prints to the console
      public static void WriteToScreen(string str)
      {
         Console.WriteLine("The String is: {0}", str);
      }
      //this method prints to a file
      public static void WriteToFile(string s)
```

```
      {
         fs = new FileStream("c:\\message.txt",
         FileMode.Append, FileAccess.Write);
         sw = new StreamWriter(fs);
         sw.WriteLine(s);
         sw.Flush();
         sw.Close();
         fs.Close();
      }
      // this method takes the delegate as parameter and uses it to
      // call the methods as required
      public static void sendString(printString ps)
      {
         ps("Hello World");
      }
      static void Main(string[] args)
      {
         printString ps1 = new printString(WriteToScreen);
         printString ps2 = new printString(WriteToFile);
         sendString(ps1);
         sendString(ps2);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
The String is: Hello World
```

CHAPTER

34

# C# Events

Events are basically a user action like key press, clicks, mouse movements, etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

## Using Delegates with Events

The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the **publisher** class. Some other class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.

A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.
A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

## Declaring Events

To declare an event inside a class, first a delegate type for the event must be declared. For example,

```
public delegate void BoilerLogHandler(string status);
```

Next, the event itself is declared, using the **event** keyword:

```
//Defining event based on the above delegate
public event BoilerLogHandler BoilerEventLog;
```

The preceding code defines a delegate named *BoilerLogHandler* and an event named *BoilerEventLog*, which invokes the delegate when it is raised.

### Example 1:

```
using System;
namespace SimpleEvent
{
   using System;

   public class EventTest
   {
      private int value;
```

```
      public delegate void NumManipulationHandler();

      public event NumManipulationHandler ChangeNum;

      protected virtual void OnNumChanged()
      {
         if (ChangeNum != null)
         {
            ChangeNum();
         }
         else
         {
            Console.WriteLine("Event fired!");
         }

      }
      public EventTest(int n )
      {
         SetValue(n);
      }
      public void SetValue(int n)
      {
         if (value != n)
         {
            value = n;
            OnNumChanged();
         }
      }
   }
   public class MainClass
   {
      public static void Main()
      {
         EventTest e = new EventTest(5);
         e.SetValue(7);
         e.SetValue(11);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Event Fired!
Event Fired!
Event Fired!
```

## Example 2:

This example provides a simple application for troubleshooting for a hot water boiler system. When the maintenance engineer inspects the boiler, the boiler temperature and pressure is automatically recorded into a log file along with the remarks of the maintenance engineer.

```
using System;
using System.IO;

namespace BoilerEventAppl
{

   // boiler class
```

```
class Boiler
{
   private int temp;
   private int pressure;
   public Boiler(int t, int p)
   {
      temp = t;
      pressure = p;
   }

   public int getTemp()
   {
      return temp;
   }
   public int getPressure()
   {
      return pressure;
   }
}
// event publisher
class DelegateBoilerEvent
{
   public delegate void BoilerLogHandler(string status);

   //Defining event based on the above delegate
   public event BoilerLogHandler BoilerEventLog;

   public void LogProcess()
   {
      string remarks = "O. K";
      Boiler b = new Boiler(100, 12);
      int t = b.getTemp();
      int p = b.getPressure();
      if(t > 150 || t < 80 || p < 12 || p > 15)
      {
         remarks = "Need Maintenance";
      }
      OnBoilerEventLog("Logging Info:\n");
      OnBoilerEventLog("Temparature " + t + "\nPressure: " + p);
      OnBoilerEventLog("\nMessage: " + remarks);
   }

   protected void OnBoilerEventLog(string message)
   {
      if (BoilerEventLog != null)
      {
         BoilerEventLog(message);
      }
   }
}
// this class keeps a provision for writing into the log file
class BoilerInfoLogger
{
   FileStream fs;
   StreamWriter sw;
   public BoilerInfoLogger(string filename)
   {
      fs = new FileStream(filename, FileMode.Append, FileAccess.Write);
      sw = new StreamWriter(fs);
   }
   public void Logger(string info)
```

```
      {
         sw.WriteLine(info);
      }
      public void Close()
      {
         sw.Close();
         fs.Close();
      }
   }
   // The event subscriber
   public class RecordBoilerInfo
   {
      static void Logger(string info)
      {
         Console.WriteLine(info);
      }//end of Logger

      static void Main(string[] args)
      {
         BoilerInfoLogger filelog = new BoilerInfoLogger("e:\\boiler.txt");
         DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();
         boilerEvent.BoilerEventLog += new
         DelegateBoilerEvent.BoilerLogHandler(Logger);
         boilerEvent.BoilerEventLog += new
         DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);
         boilerEvent.LogProcess();
         Console.ReadLine();
         filelog.Close();
      }//end of main

   }//end of RecordBoilerInfo
}
```

When the above code is compiled and executed, it produces the following result:

```
Logging info:

Temperature 100
Pressure 12

Message: O. K
```

CHAPTER

35

# C# Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

## Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace. Click the following links to check their detail.

| Class | Description and Useage |
|---|---|
| ArrayList | It represents ordered collection of an object that can be **indexed** individually.<br>It is basically an alternative to an array. However unlike array you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, add, search and sort items in the list. |
| Hashtable | It uses a **key** to access the elements in the collection.<br>A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a **key/value** pair. The key is used to access the items in the collection. |
| SortedList | It uses a **key** as well as an **index** to access the items in a list.<br><br>A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value. |
| Stack | It represents a **last-in, first out** collection of object.<br>It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called **pushing** the item and when you remove it, it is called**popping** the item. |
| Queue | It represents a **first-in, first out** collection of object.<br>It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called **enqueue** and when you remove an item, it is called**deque**. |

| | |
|---|---|
| BitArray | It represents an array of the **binary representation** using the values 1 and 0.<br>It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an **integer index**, which starts from zero. |

# ArrayList

It represents an ordered collection of an object that can be indexed individually. It is basically an alternative to an array. However unlike array you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allow dynamic memory allocation, adding, searching and sorting items in the list.

## Methods and Properties of the ArrayList Class

The following table lists some of the commonly used **properties** of the **ArrayList** class:

| Property | Description |
|---|---|
| Capacity | Gets or sets the number of elements that the ArrayList can contain. |
| Count | Gets the number of elements actually contained in the ArrayList. |
| IsFixedSize | Gets a value indicating whether the ArrayList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the ArrayList is read-only. |
| Item | Gets or sets the element at the specified index. |

The following table lists some of the commonly used **methods** of the **ArrayList** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public virtual int Add( object value );**<br>Adds an object to the end of the ArrayList. |
| 2 | **public virtual void AddRange( ICollection c );**<br>Adds the elements of an ICollection to the end of the ArrayList. |
| 3 | **public virtual void Clear();**<br>Removes all elements from the ArrayList. |
| 4 | **public virtual bool Contains( object item );**<br>Determines whether an element is in the ArrayList. |
| 5 | **public virtual ArrayList GetRange( int index, int count );**<br>Returns an ArrayList which represents a subset of the elements in the source ArrayList. |
| 6 | **public virtual int IndexOf(object);**<br>Returns the zero-based index of the first occurrence of a value in the ArrayList or in a portion of it. |
| 7 | **public virtual void Insert( int index, object value );**<br>Inserts an element into the ArrayList at the specified index. |
| 8 | **public virtual void InsertRange( int index, ICollection c );**<br>Inserts the elements of a collection into the ArrayList at the specified index. |
| 9 | **public virtual void Remove( object obj );**<br>Removes the first occurrence of a specific object from the ArrayList. |

| | |
|---|---|
| 10 | **public virtual void RemoveAt( int index );**<br>Removes the element at the specified index of the ArrayList. |
| 11 | **public virtual void RemoveRange( int index, int count );**<br>Removes a range of elements from the ArrayList. |
| 12 | **public virtual void Reverse();**<br>Reverses the order of the elements in the ArrayList. |
| 13 | **public virtual void SetRange( int index, ICollection c );**<br>Copies the elements of a collection over a range of elements in the ArrayList. |
| 14 | **public virtual void Sort();**<br>Sorts the elements in the ArrayList. |
| 15 | **public virtual void TrimToSize();**<br>Sets the capacity to the actual number of elements in the ArrayList. |

## Example:

The following example demonstrates the concept:

```
using System;
using System.Collections;

namespace CollectionApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList al = new ArrayList();

            Console.WriteLine("Adding some numbers:");
            al.Add(45);
            al.Add(78);
            al.Add(33);
            al.Add(56);
            al.Add(12);
            al.Add(23);
            al.Add(9);

            Console.WriteLine("Capacity: {0} ", al.Capacity);
            Console.WriteLine("Count: {0}", al.Count);

            Console.Write("Content: ");
            foreach (int i in al)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();
            Console.Write("Sorted Content: ");
            al.Sort();
            foreach (int i in al)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
```

```
}
```

When the above code is compiled and executed, it produces the following result:

```
Adding some numbers:
Capacity: 8
Count: 7
Content: 45 78 33 56 12 23 9
Content: 9 12 23 33 45 56 78
```

# Hashtable

The Hashtable class represents a collection of **key-and-value pairs** that are organized based on the hash code of the key. It uses the key to access the elements in the collection.
A hash table is used when you need to access elements by using **key**, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.

## Methods and Properties of the Hashtable Class

The following table lists some of the commonly used **properties** of the **Hashtable** class:

| Property | Description |
|---|---|
| Count | Gets the number of key-and-value pairs contained in the Hashtable. |
| IsFixedSize | Gets a value indicating whether the Hashtable has a fixed size. |
| IsReadOnly | Gets a value indicating whether the Hashtable is read-only. |
| Item | Gets or sets the value associated with the specified key. |
| Keys | Gets an ICollection containing the keys in the Hashtable. |
| Values | Gets an ICollection containing the values in the Hashtable. |

The following table lists some of the commonly used **methods** of the **Hashtable** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public virtual void Add( object key, object value );**<br>Adds an element with the specified key and value into the Hashtable. |
| 2 | **public virtual void Clear();**<br>Removes all elements from the Hashtable. |
| 3 | **public virtual bool ContainsKey( object key );**<br>Determines whether the Hashtable contains a specific key. |
| 4 | **public virtual bool ContainsValue( object value );**<br>Determines whether the Hashtable contains a specific value. |
| 5 | **public virtual void Remove( object key );**<br>Removes the element with the specified key from the Hashtable. |

## Example:

The following example demonstrates the concept:

```
using System;
using System.Collections;
```

```
namespace CollectionsApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Hashtable ht = new Hashtable();


         ht.Add("001", "Zara Ali");
         ht.Add("002", "Abida Rehman");
         ht.Add("003", "Joe Holzner");
         ht.Add("004", "Mausam Benazir Nur");
         ht.Add("005", "M. Amlan");
         ht.Add("006", "M. Arif");
         ht.Add("007", "Ritesh Saikia");

         if (ht.ContainsValue("Nuha Ali"))
         {
            Console.WriteLine("This student name is already in the list");
         }
         else
         {
            ht.Add("008", "Nuha Ali");
         }
         // Get a collection of the keys.
         ICollection key = ht.Keys;

         foreach (string k in key)
         {
            Console.WriteLine(k + ": " + ht[k]);
         }
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
001: Zara Ali
002: Abida Rehman
003: Joe Holzner
004: Mausam Benazir Nur
005: M. Amlan
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
```

# SortedList

The SortedList class represents a collection of key-and-value pairs that are sorted by the keys and are accessible by key and by index.

A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key, it is a Hashtable. The collection of items is always sorted by the key value.

## Methods and Properties of the SortedList Class

The following table lists some of the commonly used **properties** of the **SortedList** class:

| Property | Description |
|---|---|
| Capacity | Gets or sets the capacity of the SortedList. |
| Count | Gets the number of elements contained in the SortedList. |
| IsFixedSize | Gets a value indicating whether the SortedList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the SortedList is read-only. |
| Item | Gets and sets the value associated with a specific key in the SortedList. |
| Keys | Gets the keys in the SortedList. |
| Values | Gets the values in the SortedList. |

The following table lists some of the commonly used **methods** of the **SortedList** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public virtual void Add( object key, object value );**<br>Adds an element with the specified key and value into the SortedList. |
| 2 | **public virtual void Clear();**<br>Removes all elements from the SortedList. |
| 3 | **public virtual bool ContainsKey( object key );**<br>Determines whether the SortedList contains a specific key. |
| 4 | **public virtual bool ContainsValue( object value );**<br>Determines whether the SortedList contains a specific value. |
| 5 | **public virtual object GetByIndex( int index );**<br>Gets the value at the specified index of the SortedList. |
| 6 | **public virtual object GetKey( int index );**<br>Gets the key at the specified index of the SortedList. |
| 7 | **public virtual IList GetKeyList();**<br>Gets the keys in the SortedList. |
| 8 | **public virtual IList GetValueList();**<br>Gets the values in the SortedList. |
| 9 | **public virtual int IndexOfKey( object key );**<br>Returns the zero-based index of the specified key in the SortedList. |
| 10 | **public virtual int IndexOfValue( object value );**<br>Returns the zero-based index of the first occurrence of the specified value in the SortedList. |
| 11 | **public virtual void Remove( object key );**<br>Removes the element with the specified key from the SortedList. |
| 12 | **public virtual void RemoveAt( int index );**<br>Removes the element at the specified index of SortedList. |

| | |
|---|---|
| 13 | **public virtual void TrimToSize();**<br>Sets the capacity to the actual number of elements in the SortedList. |

## Example:

The following example demonstrates the concept:

```
using System;
using System.Collections;

namespace CollectionsApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         SortedList sl = new SortedList();

         sl.Add("001", "Zara Ali");
         sl.Add("002", "Abida Rehman");
         sl.Add("003", "Joe Holzner");
         sl.Add("004", "Mausam Benazir Nur");
         sl.Add("005", "M. Amlan");
         sl.Add("006", "M. Arif");
         sl.Add("007", "Ritesh Saikia");

         if (sl.ContainsValue("Nuha Ali"))
         {
            Console.WriteLine("This student name is already in the list");
         }
         else
         {
            sl.Add("008", "Nuha Ali");
         }

         // get a collection of the keys.
         ICollection key = sl.Keys;

         foreach (string k in key)
         {
            Console.WriteLine(k + ": " + sl[k]);
         }
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
001: Zara Ali
002: Abida Rehman
003: Joe Holzner
004: Mausam Banazir Nur
005: M. Amlan
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
```

# Stack

It represents a last-in, first out collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item.

## Methods and Properties of the Stack Class

The following table lists some of the commonly used **properties** of the **Stack** class:

| Property | Description |
|---|---|
| Count | Gets the number of elements contained in the Stack. |

The following table lists some of the commonly used **methods** of the **Stack** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public virtual void Clear();**<br>Removes all elements from the Stack. |
| 2 | **public virtual bool Contains( object obj );**<br>Determines whether an element is in the Stack. |
| 3 | **public virtual object Peek();**<br>Returns the object at the top of the Stack without removing it. |
| 4 | **public virtual object Pop();**<br>Removes and returns the object at the top of the Stack. |
| 5 | **public virtual void Push( object obj );**<br>Inserts an object at the top of the Stack. |
| 6 | **public virtual object[] ToArray();**<br>Copies the Stack to a new array. |

## Example:

The following example demonstrates use of Stack:

```
using System;
using System.Collections;

namespace CollectionsApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack st = new Stack();

            st.Push('A');
            st.Push('M');
            st.Push('G');
            st.Push('W');

            Console.WriteLine("Current stack: ");
            foreach (char c in st)
            {
                Console.Write(c + " ");
            }
```

```
            Console.WriteLine();

            st.Push('V');
            st.Push('H');
            Console.WriteLine("The next poppable value in stack: {0}",
            st.Peek());
            Console.WriteLine("Current stack: ");
            foreach (char c in st)
            {
               Console.Write(c + " ");
            }
            Console.WriteLine();

            Console.WriteLine("Removing values ");
            st.Pop();
            st.Pop();
            st.Pop();

            Console.WriteLine("Current stack: ");
            foreach (char c in st)
            {
               Console.Write(c + " ");
            }
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Current stack:
W G M A
The next poppable value in stack: H
Current stack:
H V W G M A
Removing values
Current stack:
G M A
```

# Queue

It represents a first-in, first out collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called **enqueue**, and when you remove an item, it is called **deque.**

## Methods and Properties of the Queue Class

The following table lists some of the commonly used **properties** of the **Queue** class:

| Property | Description |
|---|---|
| Count | Gets the number of elements contained in the Queue. |

The following table lists some of the commonly used **methods** of the **Queue** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public virtual void Clear();**<br>Removes all elements from the Queue. |

| | |
|---|---|
| 2 | **public virtual bool Contains( object obj );**<br>Determines whether an element is in the Queue. |
| 3 | **public virtual object Dequeue();**<br>Removes and returns the object at the beginning of the Queue. |
| 4 | **public virtual void Enqueue( object obj );**<br>Adds an object to the end of the Queue. |
| 5 | **public virtual object[] ToArray();**<br>Copies the Queue to a new array. |
| 6 | **public virtual void TrimToSize();**<br>Sets the capacity to the actual number of elements in the Queue. |

## Example:

The following example demonstrates use of Stack:

```
using System;
using System.Collections;

namespace CollectionsApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Queue q = new Queue();

         q.Enqueue('A');
         q.Enqueue('M');
         q.Enqueue('G');
         q.Enqueue('W');

         Console.WriteLine("Current queue: ");
         foreach (char c in q)
            Console.Write(c + " ");
         Console.WriteLine();
         q.Enqueue('V');
         q.Enqueue('H');
         Console.WriteLine("Current queue: ");
         foreach (char c in q)
            Console.Write(c + " ");
         Console.WriteLine();
         Console.WriteLine("Removing some values ");
         char ch = (char)q.Dequeue();
         Console.WriteLine("The removed value: {0}", ch);
         ch = (char)q.Dequeue();
         Console.WriteLine("The removed value: {0}", ch);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Current queue:
A M G W
Current queue:
A M G W V H
```

```
Removing values
The removed value: A
The removed value: M
```

# BitArray

The BitArray class manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).

It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an integer index, which starts from zero.

## Methods and Properties of the BitArray Class

The following table lists some of the commonly used **properties** of the **BitArray** class:

| Property | Description |
|---|---|
| Count | Gets the number of elements contained in the BitArray. |
| IsReadOnly | Gets a value indicating whether the BitArray is read-only. |
| Item | Gets or sets the value of the bit at a specific position in the BitArray. |
| Length | Gets or sets the number of elements in the BitArray. |

The following table lists some of the commonly used **methods** of the **BitArray** class:

| S.N | Method Name & Purpose |
|---|---|
| 1 | **public BitArray And( BitArray value );**<br>Performs the bitwise AND operation on the elements in the current BitArray against the corresponding elements in the specified BitArray. |
| 2 | **public bool Get( int index );**<br>Gets the value of the bit at a specific position in the BitArray. |
| 3 | **public BitArray Not();**<br>Inverts all the bit values in the current BitArray, so that elements set to true are changed to false, and elements set to false are changed to true. |
| 4 | **public BitArray Or( BitArray value );**<br>Performs the bitwise OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray. |
| 5 | **public void Set( int index, bool value );**<br>Sets the bit at a specific position in the BitArray to the specified value. |
| 6 | **public void SetAll( bool value );**<br>Sets all bits in the BitArray to the specified value. |
| 7 | **public BitArray Xor( BitArray value );**<br>Performs the bitwise eXclusive OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray. |

## Example:

The following example demonstrates the use of BitArray class:

```
using System;
using System.Collections;

namespace CollectionsApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            //creating two  bit arrays of size 8
            BitArray ba1 = new BitArray(8);
            BitArray ba2 = new BitArray(8);
            byte[] a = { 60 };
            byte[] b = { 13 };

            //storing the values 60, and 13 into the bit arrays
            ba1 = new BitArray(a);
            ba2 = new BitArray(b);

            //content of ba1
            Console.WriteLine("Bit array ba1: 60");
            for (int i = 0; i < ba1.Count; i++)
            {
                Console.Write("{0, -6} ", ba1[i]);
            }
            Console.WriteLine();

            //content of ba2
            Console.WriteLine("Bit array ba2: 13");
            for (int i = 0; i < ba2.Count; i++)
            {
                Console.Write("{0, -6} ", ba2[i]);
            }
            Console.WriteLine();


            BitArray ba3 = new BitArray(8);
            ba3 = ba1.And(ba2);

            //content of ba3
            Console.WriteLine("Bit array ba3 after AND operation: 12");
            for (int i = 0; i < ba3.Count; i++)
            {
                Console.Write("{0, -6} ", ba3[i]);
            }
            Console.WriteLine();

            ba3 = ba1.Or(ba2);
            //content of ba3
            Console.WriteLine("Bit array ba3 after OR operation: 61");
            for (int i = 0; i < ba3.Count; i++)
            {
                Console.Write("{0, -6} ", ba3[i]);
            }
            Console.WriteLine();

            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Bit array ba1: 60
False False True True True True False False
Bit array ba2: 13
True False True True False False False False
Bit array ba3 after AND operation: 12
False False True True False False False False
Bit array ba3 after OR operation: 61
True False True True False False False False
```

CHAPTER

36

# C# Generics

Generics allow you to delay the specification of the data type of programming elements in a class or a method until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept:

```
using System;
using System.Collections.Generic;

namespace GenericApplication
{
    public class MyGenericArray<T>
    {
        private T[] array;
        public MyGenericArray(int size)
        {
            array = new T[size + 1];
        }
        public T getItem(int index)
        {
            return array[index];
        }
        public void setItem(int index, T value)
        {
            array[index] = value;
        }
    }

    class Tester
    {
        static void Main(string[] args)
        {
            //declaring an int array
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);
            //setting values
            for (int c = 0; c < 5; c++)
            {
                intArray.setItem(c, c*5);
            }
```

```
            //retrieving the values
            for (int c = 0; c < 5; c++)
            {
                Console.Write(intArray.getItem(c) + " ");
            }
            Console.WriteLine();
            //declaring a character array
            MyGenericArray<char> charArray = new MyGenericArray<char>(5);
            //setting values
            for (int c = 0; c < 5; c++)
            {
                charArray.setItem(c, (char)(c+97));
            }
            //retrieving the values
            for (int c = 0; c< 5; c++)
            {
                Console.Write(charArray.getItem(c) + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
0 5 10 15 20
a b c d e
```

## Features of Generics

Using generics is a technique that enriches your programs in the following ways:

- It helps you to maximize code reuse, type safety, and performance.
- You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in the *System.Collections.Generic* namespace. You may use these generic collection classes instead of the collection classes in the *System.Collections*namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- You may create generic classes constrained to enable access to methods on particular data types.
- You may get information on the types used in a generic data type at run-time by means of reflection.

## Generic Methods

In the previous example, we have used a generic class; we can declare a generic method with a type parameter. The following program illustrates the concept:

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl
{
    class Program
    {
```

```
        static void Swap<T>(ref T lhs, ref T rhs)
        {
            T temp;
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
        static void Main(string[] args)
        {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            //display values before swap:
            Console.WriteLine("Int values before calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            //call swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            //display values after swap:
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

# Generic Delegates

You can define a generic delegate with type parameters. For example:

```
delegate T NumberChanger<T>(T n);
```

The following example shows use of this delegate:

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl
```

```
{
    class TestDelegate
    {
        static int num = 10;
        public static int AddNum(int p)
        {
            num += p;
            return num;
        }

        public static int MultNum(int q)
        {
            num *= q;
            return num;
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            //create delegate instances
            NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
            NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);
            //calling the methods using the delegate objects
            nc1(25);
            Console.WriteLine("Value of Num: {0}", getNum());
            nc2(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of Num: 35
Value of Num: 175
```

CHAPTER

37

# C# Anonymous Methods

We discussed that delegates are used to reference any methods that has the same signature as that of the delegate. In other words, you can call a method that can be referenced by a delegate using that delegate object.

**Anonymous methods** provide a technique to pass a code block as a delegate parameter. Anonymous methods are basically methods without a name, just the body.

You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

## Syntax for Writing an Anonymous Method

Anonymous methods are declared with the creation of the delegate instance, with a **delegate** keyword. For example,

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x)
{
    Console.WriteLine("Anonymous Method: {0}", x);
};
```

The code block *Console.WriteLine("Anonymous Method: {0}", x);* is the body of the anonymous method.

The delegate could be called both with anonymous methods as well as named methods in the same way, i.e., by passing the method parameters to the delegate object.

For example,

```
nc(10);
```

### Example:

The following example demonstrates the concept:

```
using System;

delegate void NumberChanger(int n);
namespace DelegateAppl
{
    class TestDelegate
    {
```

```
        static int num = 10;
        public static void AddNum(int p)
        {
            num += p;
            Console.WriteLine("Named Method: {0}", num);
        }

        public static void MultNum(int q)
        {
            num *= q;
            Console.WriteLine("Named Method: {0}", num);
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            //create delegate instances using anonymous method
            NumberChanger nc = delegate(int x)
            {
               Console.WriteLine("Anonymous Method: {0}", x);
            };

            //calling the delegate using the anonymous method
            nc(10);

            //instantiating the delegate using the named methods
            nc =  new NumberChanger(AddNum);

            //calling the delegate using the named methods
            nc(5);

            //instantiating the delegate using another named methods
            nc =  new NumberChanger(MultNum);

            //calling the delegate using the named methods
            nc(2);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Anonymous Method: 10
Named Method: 15
Named Method: 30
```

CHAPTER

38

# C# Unsafe Codes

C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier. The **unsafe code** or the unmanaged code is a code block that uses a **pointer** variable.

## Pointer Variables

A **pointer** is a variable whose value is the address of another variable i.e., the direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address.

The general form of a pointer variable declaration is:

```
type *var-name;
```

Following are valid pointer declarations:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The following example illustrates use of pointers in C#, using the unsafe modifier:

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ",  var);
```

```
            Console.WriteLine("Address is: {0}",  (int)p);

            Console.ReadKey();

        }

    }

}
```

When the above code wass compiled and executed, it produces the following result:

```
Data is: 20
Address is: 99215364
```

Instead of declaring an entire method as unsafe, you can also declare a part of the code as unsafe. The example in the following section shows this.

# Retrieving the Data Value Using a Pointer

You can retrieve the data stored at the located referenced by the pointer variable, using the **ToString()**method. Following example demonstrates this:

```
using System;

namespace UnsafeCodeApplication

{

   class Program

   {

      public static void Main()

      {

         unsafe

         {

            int var = 20;

            int* p = &var;

            Console.WriteLine("Data is: {0} " , var);

            Console.WriteLine("Data is: {0} " , p->ToString());

            Console.WriteLine("Address is: {0} " , (int)p);

         }

         Console.ReadKey();

      }

   }

}
```

When the above code was compiled and executed, it produces the following result:

```
Data is: 20
Data is: 20
Address is: 77128984
```

## Passing Pointers as Parameters to Methods

You can pass a pointer variable to a method as parameter. The following example illustrates this:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

# Accessing Array Elements Using a Pointer

In C#, an array name and a pointer to a data type same as the array data, are not the same variable type. For example, int *p and int[] p, are not same type. You can increment the pointer variable p because it is not fixed in memory but an array address is fixed in memory, and you can't increment that.

Therefore, if you need to access an array data using a pointer variable, as we traditionally do in C, or C++ ( please check: C Pointers), you need to fix the pointer using the **fixed** keyword.

The following example demonstrates this:

```
using System;

namespace UnsafeCodeApplication

{

   class TestPointer

   {

      public unsafe static void Main()

      {

         int[]  list = {10, 100, 200};

         fixed(int *ptr = list)


         /* let us have array address in pointer */

         for ( int i = 0; i < 3; i++)

         {

            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));

            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));

         }

         Console.ReadKey();

      }

   }

}
```

When the above code was compiled and executed, it produces the following result:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
```

```
Value of list[2] = 200
```

# Compiling Unsafe Code

For compiling unsafe code, you have to specify the **/unsafe** command-line switch with command-line compiler.

For example, to compile a program named prog1.cs containing unsafe code, from command line, give the command:

```
csc /unsafe prog1.cs
```

If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.

To do this:

- Open **project properties** by double clicking the properties node in the Solution Explorer.
- Click on the **Build** tab.
- Select the option "**Allow unsafe code**".

CHAPTER

39

# C# Multithreading

A thread is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations then it is often helpful to set different execution paths or threads, with each thread performing a particular job.

Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

So far we have written programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

## Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread:

- **The Unstarted State**: it is the situation when the instance of the thread is created but the Start method has not been called.
- **The Ready State**: it is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State**: a thread is not runnable, when:
    - Sleep method has been called
    - Wait method has been called
    - Blocked by I/O operations
- **The Dead State**: it is the situation when the thread has completed execution or has been aborted.

## The Main Thread

In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.

The following program demonstrates main thread execution:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class MainThreadProgram
    {
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}", th.Name);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
This is MainThread
```

# Commonly Used Properties and Methods of the Thread Class

The following table shows some of the most commonly used **properties** of the **Thread** class:

| Property | Description |
|---|---|
| CurrentContext | Gets the current context in which the thread is executing. |
| CurrentCulture | Gets or sets the culture for the current thread. |
| CurrentPrinciple | Gets or sets the thread's current principal (for role-based security). |
| CurrentThread | Gets the currently running thread. |
| CurrentUICulture | Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time. |
| ExecutionContext | Gets an ExecutionContext object that contains information about the various contexts of the current thread. |
| IsAlive | Gets a value indicating the execution status of the current thread. |
| IsBackground | Gets or sets a value indicating whether or not a thread is a background thread. |
| IsThreadPoolThread | Gets a value indicating whether or not a thread belongs to the managed thread pool. |
| ManagedThreadId | Gets a unique identifier for the current managed thread. |
| Name | Gets or sets the name of the thread. |
| Priority | Gets or sets a value indicating the scheduling priority of a thread. |

| | |
|---|---|
| ThreadState | Gets a value containing the states of the current thread. |

The following table shows some of the most commonly used **methods** of the **Thread** class:

| S.N | Method Name & Description |
|---|---|
| 1 | **public void Abort()**<br>Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread. |
| 2 | **public static LocalDataStoreSlot AllocateDataSlot()**<br>Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead. |
| 3 | **public static LocalDataStoreSlot AllocateNamedDataSlot( string name)**<br>Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead. |
| 4 | **public static void BeginCriticalRegion()**<br>Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain. |
| 5 | **public static void BeginThreadAffinity()**<br>Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread. |
| 6 | **public static void EndCriticalRegion()**<br>Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task. |
| 7 | **public static void EndThreadAffinity()**<br>Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread. |
| 8 | **public static void FreeNamedDataSlot(string name)**<br>Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead. |
| 9 | **public static Object GetData( LocalDataStoreSlot slot )**<br>Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead. |
| 10 | **public static AppDomain GetDomain()**<br>Returns the current domain in which the current thread is running. |
| 11 | **public static AppDomain GetDomain()**<br>Returns a unique application domain identifier |
| 12 | **public static LocalDataStoreSlot GetNamedDataSlot( string name )**<br>Looks up a named data slot. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead. |
| 13 | **public void Interrupt()**<br>Interrupts a thread that is in the WaitSleepJoin thread state. |
| 14 | **public void Join()**<br>Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping. This method has different overloaded forms. |
| 15 | **public static void MemoryBarrier()**<br>Synchronizes memory access as follows: The processor executing the current thread cannot reorder |

| | |
|---|---|
| | instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier. |
| 16 | **public static void ResetAbort()**<br>Cancels an Abort requested for the current thread. |
| 17 | **public static void SetData( LocalDataStoreSlot slot, Object data )**<br>Sets the data in the specified slot on the currently running thread, for that thread's current domain. For better performance, use fields marked with the ThreadStaticAttribute attribute instead. |
| 18 | **public void Start()**<br>Starts a thread. |
| 19 | **public static void Sleep( int millisecondsTimeout )**<br>Makes the thread pause for a period of time. |
| 20 | **public static void SpinWait( int iterations )**<br>Causes a thread to wait the number of times defined by the iterations parameter |
| 21 | **public static byte VolatileRead( ref byte address )**<br>**public static double VolatileRead( ref double address )**<br>**public static int VolatileRead( ref int address )**<br>**public static Object VolatileRead( ref Object address )**<br>Reads the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method has different overloaded forms. Only some are given above. |
| 22 | **public static void VolatileWrite( ref byte address, byte value )**<br>**public static void VolatileWrite( ref double address, double value )**<br>**public static void VolatileWrite( ref int address, int value )**<br>**public static void VolatileWrite( ref Object address, Object value )**<br>Writes a value to a field immediately, so that the value is visible to all processors in the computer. This method has different overloaded forms. Only some are given above. |
| 23 | **public static bool Yield()**<br>Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to. |

# Creating Threads

Threads are created by extending the Thread class. The extended Thread class then calls the **Start()**method to begin the child thread execution.

The following program demonstrates the concept:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
```

```
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
In Main: Creating the Child thread
Child thread starts
```

# Managing Threads

The Thread class provides various methods for managing threads.

The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time.

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
            // the thread is paused for 5000 milliseconds
            int sleepfor = 5000;
            Console.WriteLine("Child Thread Paused for {0} seconds",
                              sleepfor / 1000);
            Thread.Sleep(sleepfor);
            Console.WriteLine("Child thread resumes");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

# Destroying Threads

The **Abort()** method is used for destroying threads.
The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this:

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            try
            {

                Console.WriteLine("Child thread starts");
                // do some work, like counting to 10
                for (int counter = 0; counter <= 10; counter++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine(counter);
                }
                Console.WriteLine("Child Thread Completed");

            }
            catch (ThreadAbortException e)
            {
                Console.WriteLine("Thread Abort Exception");
            }
            finally
            {
                Console.WriteLine("Couldn't catch the Thread Exception");
            }

        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            //stop the main thread for some time
            Thread.Sleep(2000);
            //now abort the child
            Console.WriteLine("In Main: Aborting the Child thread");
            childThread.Abort();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
In Main: Creating the Child thread
```

```
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```